Change Management for Distributed Ontologies

Klein, M.C.A.

2004

Change Management for Distributed Ontologies

Michel Klein

SIKS Dissertation Series No. 2004-11.

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Graduate School for Information and Knowledge Systems.

Promotiecommissie:
prof.dr. A.Th. Schreiber (promotor)
prof.dr. J. M. Akkermans (promotor)
prof.dr. D.A. Fensel (University of Innsbruck, Austria)
prof.dr. F. van Harmelen (Vrije Universiteit Amsterdam)
dr. Steffen Staab (University of Karlsruhe, Germany)
prof.dr. H. van Vliet (Vrije Universiteit Amsterdam)
prof.dr. B. J. Wielinga (Universiteit van Amsterdam)
prof.dr. R. J. Wieringa (Technische Universiteit Twente)

ISBN 90-9018400-7

VRIJE UNIVERSITEIT

Change Management for Distributed Ontologies

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op dinsdag 14 september 2004 om 13.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Michel Christiaan Alexander Klein

geboren te Purmerend

promotoren: prof.dr. A.Th. Schreiber
prof.dr. J.M. Akkermans

Preface

A writer keeps surprising himself...
he doesn't know what he is saying
until he sees it on the page.
— Thomas Williams

It is often said that the preface is the most read part of a thesis. If this is true, what is the contribution of distributing a thesis to the progress of science? Why does NWO[1] pay for printing it?

Let us first try to answer the question *why* the preface is thought to the most read part of a thesis. A possible answer is that the preface is the only part of thesis that is understandable for outsiders. If we assume that a preface normally doesn't describe the scientific contribution, this would be a bad sign for scientists about their ability to explain their achievements.

However, I doubt whether this is the genuine reason. In this thesis, for example, the summary (page 181) is probably easier to understand than the preface with its numerous references to people that many the readers will not know. My hypothesis is that the preface is among the most read parts of a thesis because it gives a glimpse of the *real* process behind the nicely shaped-up story in the book. It hints about the moments of disappointment and desperateness, the time lost in mastering LaTeX, the writer-blocks, the procrastinating, the "essential" philosophical discussions with colleagues, and the oh-so-nice but not very relevant programming tasks. In the end, a preface shows that doing research is not the well-structured process that it seems to be, but that it is as humble as all other activities. It may very well be that this message is the essence of what scientists have to say to the world...

Therefore, for the sake of science, continue reading the preface and don't feel obliged to read further.

The research described in this thesis has been performed within the KIRMIT project. This thesis is probably not the outcome that the people who wrote the proposal for the KIRMIT project had in mind. When the project started, we expected to work with two PhD students on multimedia indexing and retrieval, but because of various reasons things went different. I would like to thank Bob Wielinga, Bert Bredeweg and Hans Akkermans for the freedom and the time they gave me to find my own way. Hans has been very

[1] The Dutch National Organization for Scientific Research.

inspiring during the whole process, with his clear ideas about research methodologies and his talent to bring difficult issues back to their essence.

I'm very grateful to Guus Schreiber for his enormous help in the last year. He spend a lot of time reading all my writings and helped me to transform my scattered work into coherent thesis. His detailed comments and questions helped me to understand what I tried to write down (seeing it on a page was not yet sufficient for me).

Dieter also had a big influence on my research process. I would like to thank him for pushing me to the frontiers, pointing out fruitful research directions and for making many things possible. For example, he arranged a stay at Stanford University for me. The three months in the USA have been very enjoyable, both from a scientific as from a personal perspective. I would like to thank Natasha Noy and Mark Musen for their willingness to host me and the collaboration that we have.

Furthermore, I would like to acknowledge the collaboration with the OntoText company in Bulgaria. Especially I would like to thank Naso and Damyan for their contributions and the effort they put into our collaboration.

The interactions and discussions with colleagues have been most teachable, and this is what colored the daily practice. I would like to thank Guido for the first years we spent together in the office, when both of us had to find our way as a PhD student, and for the talks later on when we were moved to different floors.

Of course, I have to mention Borys. I can hardly imagine a person that is more different from me than he. Once, we were accused of fighting like a couple that is together for too long. However, when looking back I have to conclude that sharing an office with him has been very nice. He made me realize that the center of Europe is much further to the east than what I was used to think. Besides all practical assistance that we gave to each other, I enjoyed the numerous discussions about the meaning of science, research and even the meaning of love and life. I think that we even agreed once on some existential issue, but I can't remember anymore which one it was.

Then, I would like to thank all other colleagues. The 12 o'clock lunch group that in the end only consisted of Arno, Michel and Martijn, and the colleagues from the BI and AI groups. I'm especially grateful to Borys, Jeen, Laura, Mark, Marta and Radu for proofreading chapters of my thesis, and to Joost for his advices about cover.

Finally, I would like to thank Linda. Linda, your contribution is from a completely different kind, but has been essential. I thank you for your love, support and patience. Sharing my life with you is worth more than whatever I can imagine. And yes, you are still 2750 copies ahead of me.

Michel Klein, Bussum, July 2004.

Contents

Chapter 1

Introduction

For the times they are a-changin'.
— Bob Dylan, 1963

That which has been is that which shall be;
and that which has been done is that which shall be done:
and there is no new thing under the sun.
— Ecclesiastes 1:9, ± 300 B.C.

Change is a constant and important factor in human history. Many examples can be given. Changes inspire people to think, write, talk, sing or act: Bob Dylan in 1963, Joop den Uyl[1] in 1973, George W. Bush in 2001, they all claimed that the world had changed. The outline of the history is marked with the major changes. Changes are seen as the milestones on the road of progress. The major political division in western countries is still the one between those who want to preserve what they think is good, and those who want to change to what they think is better. Scientists are hunting for insights that change the view on the world.

This thesis is about change, although in a much more restricted domain. We consider the change of artificial constructs, called ontologies, that are used by computers for handling information.

In the computer science jargon, an ontology is a formal specification of a particular view on the important concepts within a respective domain (Gruber, 1993). That is, it gives a formal description of the "things that exist" in a particular subject area. Typically, an ontology consists of a hierarchy of concepts with a specification of their characteristics and relationships. The idea behind applying ontologies to information management is that computers can exploit the knowledge that is contained in an ontology to handle information in a way that is similar to what a human who shares the same world view would do. In this sense, ontologies enable information sharing between humans and computers.

This approach is getting more attention with the rise of the Semantic Web (Berners-Lee et al., 2001). In this, up to now highly academic, extension of the World Wide Web,

[1] Dutch prime minister from 1973 till 1977

computers use explicit descriptions of the meaning of data on web pages to handle it in a more intelligent way. This in contrast with the current implementation of the Web, where computers mainly *transport* and *display* web pages. In the Semantic Web scenario, the information *inside* pages is accessed and used by computers. There are basically two techniques that underly the Semantic Web. First, data on pages is structured and represented in such a way that it can be accessed by computers at a fine-grained level. Second, these pieces of data are related to concepts that are defined by ontologies. The definitions in the ontologies give computers knowledge about how to use and combine the information.

However, neither the data on the web, nor the ontologies themselves are permanent and stable. Data on web pages can change because it encodes temporal facts, e.g. the weather forecast or stock quotes, or for many other reasons. Ontologies can change as well, for example because the view on the world changes (e.g. what are the major political issues), or because the knowledge about certain topics improves (e.g. the effect of a specific drug on a disease), or because the world itself changes (e.g. which countries exist at the Balkan).

Ontology changes are important to consider because they have effect on the way data should interpreted and handled. However, *which* effect changes have can not be determined by looking at the ontologies on their own. This also depends on the reasons behind a change and the specific task for which an ontology is used. Moreover, when several people make changes to ontologies, questions arise about identification of temporary world views, versioning of the specifications of these views, and so on.

How to effectively use ontologies for computerized information management is still an ongoing research issue (Stuckenschmidt, 2003). Applying this in a dynamic environment where ontologies change over time is even more a challenge. A better understanding of the problem area and methods that take the problems sketched above into account are needed.

There are other research areas that also consider change of information sources. Most closely related is the area of evolving database schemata (Roddick, 1995; Roddick et al., 2000), and especially evolving object oriented databases (Banerjee et al., 1987). Although from the surface this area look very similar ontology change, the scale and extent of the problems around ontology change justify a consideration on its own. We compare database schema evolution with ontology evolution in Chapter 3 of this thesis.

A specific subfield of database evolution is version modelling (Katz, 1990; Klahold et al., 1986). This field provides concepts for structuring databases that evolve over time. General ideas from this area, like change propagation and modeling the derivation relations, can be used for ontology change evolution as well and have found their way into the framework described in Chapter 4.

Another area that addresses change of information sources is software engineering, and especially software configuration management. This field studies the standards and procedures for managing an evolving system product (Sommerville, 2001). This includes issues such as multiple developers working on the same code at the same time, targeting multiple platforms, supporting multiple versions, and controlling the status of code. In principle, these issues are also relevant for collaborative ontology development; however because we assume an uncontrolled evolution process, we do not discuss these topics

in this thesis. Note that in the context of software engineering, *change management* refers to the structured review process for proposed changes to software. In this thesis, however, the term change management is used in a broader sense: it refers to the process of performing the changes as well as to the process of coping with the consequences of changes.

1.1 Research Question

The central research question in this thesis is the following:

> "Which mechanisms and methods are required to cope with ontology change in a dynamic and distributed setting, where ontologies are used as means to improve computerized information exchange?"

This general question can be detailed into three smaller questions:

1. What are the specific characteristics of change management for distributed ontologies?
2. What is an adequate representation of changes between ontologies?
3. What methods and techniques can be developed to solve possible problems caused by ontology change?

1.2 Approach

In this thesis, we try to achieve a better understanding of a complex problem. We develop the understanding by analyzing the context of the problem and comparing it with problems and solutions in related areas. Based on this, we introduce a framework that relates important items in the problem area. Then the framework is used to explore a number of techniques that could help solving some of the problem in certain situations. The techniques follow from a theoretical analysis of the problem and its context.

The current status of the Semantic Web makes it very difficult to evaluate these techniques in a realistic setting. Up to now, the Semantic Web is not much more than a vision. Although a number of techniques are in place and some tools and applications have been developed, the critical mass of structured data and related ontologies is still missing. A thorough evaluation of the proposed framework and the proposed techniques is therefore not possible.

To provide some evidence of the usability of the framework, we implement parts of it in automated tools and we conduct a number of theoretical and practical studies. These tools and studies do not aim at proving the framework as a whole, but show the technical correctness and / or feasibility of the techniques that we propose.

We apply a number of different research methods in this thesis. For the comparison with related areas, we perform a literature survey. An overview of current change management is based on five semi-structured interviews. We explore some of the developed

techniques in the framework by prototyping computerized tools. The practical studies consist of two case studies and a user study.

1.3 Contributions and Outline

The main contribution of this thesis is a better understanding of the problem of ontology evolution. In developing this understanding and as elements of it, we make a number of other contributions as well.

The first part of the thesis describes the context in which the research is conducted and the requirements for an ontology change management solution.

In **Chapter 2**, we describe the major representation languages that are used on the Semantic Web and we explain their distinguished role. In the last part of the chapter, we present a general principle for expressing a knowledge representation language within the Resource Description Framework (a general metadata representation mechanism designed for the web). This principle has become the basis for the representation of newer ontology languages for the Semantic Web as well.

In **Chapter 3**, we derive the requirements for ontology change management. We do this by analyzing current change management practices for four large centralized ontologies. Also, we compare ontology change management with the database schema versioning. The chapter results in a wish list for ontology change management.

The second part of the thesis describes the actual framework for ontology change management that we propose.

Chapter 4 describes the assumptions, the vocabulary and the elements of the framework. It also briefly sketches how the elements of the framework can interact to solve particular problems. This chapter gives the general picture of the framework, whereas different elements of it are detailed in the next two chapters.

The main contribution of **Chapter 5** is a mechanism to represent changes between ontologies. An central element in this representation is a taxonomy, i.e. an "ontology", of change operations that can be performed on an ontology. We produce this with help of the meta model of an ontology language. To generalize from one specific ontology language, we do this for two different ontology languages, namely OWL and OKBC. We compare both meta models to assess the generality of a change language for OWL.

Chapter 6 shows how the framework can be used to solve different ontology evolution problems. We present a method to use the framework for data interpretation via different versions of ontologies. We also introduce a computational cheap approach for maintaining the integrity of mappings between ontologies. In addition, we explain how the framework can be applied for change visualization and for synchronization of different ontology versions.

In the third part of the thesis, we apply the framework that we have developed.

Chapter 7 describes a three tools that implement some of the mechanisms. We discuss the OntoView system, which implements a comparison mechanism for ontologies, and we describe an extension to the PROMPTdiff tool (Noy and Musen, 2002), which is able export differences between ontologies in the change representation mechanisms that

we introduced in chapter 6. We also show a tool that uses the framework to improve the visualization of ontology changes.

In **Chapter 8** we perform two theoretical studies and one practical study to assess the feasibility of the framework. In the first study, we use a large set of ontologies that are evolved out of each other to show that the change representation mechanism is complete enough to cover a realistic ontology evolution scenario. The second study is performed in the context of the WonderWeb project[2]. We create mappings from an artificial ontology to an external ontology based on a database. When this external ontology evolves because of the application of a cleaning methodology, we use our framework to predict the effect on the integrity of the mappings. This study illustrates how the mechanism described in Chapter 6 can work in practice. Finally, **Chapter 9** concludes the thesis and looks forward.

1.4 Publications

"Zoals je ziet (...) sleurt de wetenschap
haar bescheiden dienaren niet naar
de beroerdste plekken op aarde."
— De Procedure, Harry Mulisch.

Parts of this theses have been published before.

- Chapter 2 on ontology languages is based on two publications. The explanation of the respective languages is published as "Klein, M. (2001b). XML, RDF, and Relatives (short tutorial). *IEEE Intelligent Systems, special issue on "Semantic Web Technology"*, 16(2):26–28". The explanation of expressing other languages in RDF is published as "Broekstra, J., Klein, M., Decker, S., Fensel, D., van Harmelen, F., and Horrocks, I. (2002b). Enabling knowledge representation on the Web by extending RDF Schema. *Computer Networks*, 39(5):609–634"
- The first part of Chapter 3 is published as "Klein, M. (2001a). Combining and relating ontologies: an analysis of problems and solutions. In Gomez-Perez, A., Gruninger, M., Stuckenschmidt, H., and Uschold, M., editors, *Workshop on Ontologies and Information Sharing, IJCAI'01*, Seattle, USA". The second section of the chapter is based on "Noy, N. F. and Klein, M. (2004). Ontology evolution: Not the same as schema evolution. *Knowledge and Information Systems*, 6(4):428–440".
- A preliminary description of the framework in chapter 4 is published as "Klein, M. and Noy, N. F. (2003). A component-based framework for ontology evolution. In *Proceedings of the Workshop on Ontologies and Distributed Systemsm, IJCAI '03*, Acapulco, Mexico. Also available as Technical Report IR-504, Vrije Universiteit Amsterdam".
- Parts of Chapter 6 are published as "Stuckenschmidt, H. and Klein, M. (2003). Integrity and change in modular ontologies. In *Proceedingso of the 18th International Joint Conference on Artificial Intelligence*, Acapulco, Mexico".

[2]EU IST project, see `http://wonderweb.semanticweb.org`.

- The tools that are presented in chapter 7 are also described in “Klein, M., Fensel, D., Kiryakov, A., and Ognyanov, D. (2002a). Ontology versioning and change detection on the web. In *13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02)*, number 2473 in LNCS, page 197 ff, Sigüenza, Spain”, “Klein, M., Kiryakov, A., Ognyanov, D., and Fensel, D. (2002b). Finding and characterizing changes in ontologies. In *Proceedings of the 21st International Conference on Conceptual Modeling (ER2002)*, number 2503 in LNCS, pages 79–89, Tampere, Finland”, and “Noy, N. F. and Klein, M. (2003). Visualizing changes during ontology evolution. In *Collected Posters ISWC 2003*, Sanibal Island, Florida, USA”.
- Finally, the case study from chapter 8 is published as “Klein, M. and Stuckenschmidt, H. (2003). Evolution management for interconnected ontologies. In *Workshop on Semantic Integration at ISWC 2003*, Sanibel Island, Florida”.

Part I

Context and Requirements

Chapter 2

Languages for the Semantic Web

***Note**: This chapter is based on two publications. The description of XML, RDF and RDFS in Section 2.1 has been published as tutorial in* IEEE Intelligent Systems *(Klein, 2001b). The mechanism to represent an ontology language as extension of RDFS (Section 2.2) is published in the journal of* Computer Networks *(Broekstra et al., 2002b), and is co-authored by Jeen Broekstra, Stefan Decker, Dieter Fensel, Frank van Harmelen and Ian Horrocks. The description of the OIL language (Section 2.1.5) is taken from the same publication.*

Languages for representing data and knowledge are an important element of the Semantic Web. This chapter gives an introduction to the most important languages, namely XML, RDF, RDF Schema and OIL and its successors DAML+OIL and OWL. In the first part of the chapter, we describe the basic principles of the different formalisms and explain their role. The second part of the chapter describes a general mechanism that can be used to encode a more expressive knowledge representation scheme as extension to RDF Schema. We demonstrate how we have done this for the OIL language, but the mechanism can be used for other languages as well.

2.1 Main Languages

There many different computer languages that play a role in the Semantic Web. Most languages are based on XML or use XML as syntax; some have connections to RDF or RDF Schema. In this section, we will briefly introduce XML, XML Schema, RDF, RDF Schema, and OIL.

2.1.1 XML as a basis

XML (eXtensible Markup Language) is a specification for computer-readable documents. Markup means that certain sequences of characters in the document contain information indicating the role of the document's content. The markup describes the

document's data layout and logical structure and makes the information self-describing, in a sense. It takes the form of words between angle brackets, called tags—for example, `<name>` or `<h1>`. In this aspect, XML looks very much like the well-known language HTML.

However, *extensible* indicates an important difference and a main characteristic of XML. XML is actually a *metalanguage*: a mechanism for representing other languages in a standardized way. In other words, XML only provides a data format for structured documents, without specifying an actual vocabulary. This makes XML universally applicable: you can define customized markup languages for unlimited types of documents. This has already occurred on a massive scale. Besides many proprietary languages—ranging from electronic order forms to application file formats—a number of standard languages are defined in XML (called XML applications). For example, XHTML is a redefinition of HTML 4.0 in XML.

Let's take a more detailed look at XML. The main markup entities in XML are elements. They consist normally of an opening tag and a closing tag—for example, `<person>` and `</person>`. Elements might contain other elements or text. If an element has no content, it can be abbreviated as `<person/>`. Elements should be properly nested: a child element's opening and closing tags must be within its parent's opening and closing tags. Every XML document must have exactly one root element. Elements can carry attributes with values, encoded as additional "word = value" pairs inside an element tag—for example, `<person name="John">`. Here is a piece of XML:

```
<?xml version="1.0"?>
<employees> List of persons in company:
  <person name="John">
    <phone>47782</phone>
    On leave for 2003.
  </person>
</employees>
```

XML does not imply a specific interpretation of the data. Of course, on account of the tag's names, the meaning of the previous piece of XML seems obvious to human users, but it is not formally specified! The only legitimate interpretation is that XML code contains named entities with sub-entities and values; that is, every XML document forms an ordered, labeled tree. This generality is both XML's strength and its weakness. You can encode all kinds of data structures in an unambiguous syntax, but XML does not specify the data's use and semantics. The parties that use XML for their data exchange must agree beforehand on the vocabulary, its use, and its meaning.

2.1.2 DTDs and XML Schemas

Such an agreement can be partly specified by *Document Type Definitions* and XML Schemas. Although DTDs and XML Schemas do not specify the data's meaning, they do specify the names of elements and attributes (the vocabulary) and their use in documents. Both are mechanisms with which you can specify the structure of XML documents. You can then validate specific documents against the structure prescription specified by a DTD or an XML Schema.

DTDs provide only a simple structure prescription: they specify the allowed nesting of elements, the elements' possible attributes, and the locations where normal text is allowed. For example, a DTD might prescribe that every person element must have a name attribute and may have a child element called phone whose content must be text. A DTD's syntax looks a bit awkward, but it is actually quite simple.

XML Schemas are a proposed successor to DTDs. The XML Schema definition is still a candidate recommendation from the W3C (World Wide Web Consortium), which means that, although it is considered stable, it might still undergo small revisions. XML Schemas have several advantages over DTDs. First, the XML Schema mechanism provides a richer grammar for prescribing the structure of elements. For example, you can specify the exact number of allowed occurrences of child elements, you can specify default values, and you can put elements in a choice group, which means that exactly one of the elements in that group is allowed at a specific location. Second, it provides data typing. In the example in the previous paragraph, you could prescribe the phone element's content as five digits, possibly preceded by another five digits between brackets. A third advantage is that the XML Schema definition provides inclusion and derivation mechanisms. This lets you reuse common element definitions and adapt existing definitions to new practices.

A final difference from DTDs is that XML Schema prescriptions use XML as their encoding syntax. (XML is a metalanguage, remember?) This simplifies tool development, because both the structure prescription and the prescribed documents use the same syntax. The XML Schema specification's developers exploited this feature by using an XML Schema document to define the class of XML Schema documents. After all, because an XML Schema prescription is an XML application, it must obey rules for its structure, which can be defined by another XML Schema prescription. However, this recursive definition can be a bit confusing.

2.1.3 Resource Description Format

XML provides a syntax to encode data; the Resource Description Framework is a mechanism to tell something about data. As its name indicates, it is not a language but a model for representing data about "things on the Web." This type of data about data is called metadata. The "things" are resources in RDF vocabulary.

RDF's basic data model is simple: besides resources, it contains properties and statements. A property is a specific aspect, characteristic, attribute, or relation that describes a resource. A statement consists of a specific resource with a named property plus that property's value for that resource. This value can be another resource or a literal value: free text, basically. Altogether, an RDF description is a list of triples: an object (a resource), an attribute (a property), and a value (a resource or free text). For example, to state that a specific Web page was created by something with a name "John" and a phone number "47782", the following three triples are required.

subject	**predicate**	**object**
http://www.w3.org/	created_by	*anon_1*
anon_1	name	"John"
anon_1	phone_number	"47782"

You can easily depict an RDF model as a directed labeled graph. To do this, you draw an oval for every resource and an arrow for every property, and you represent literal values as boxes with values. Figure 2.1 shows such a graph for the triples listed above.

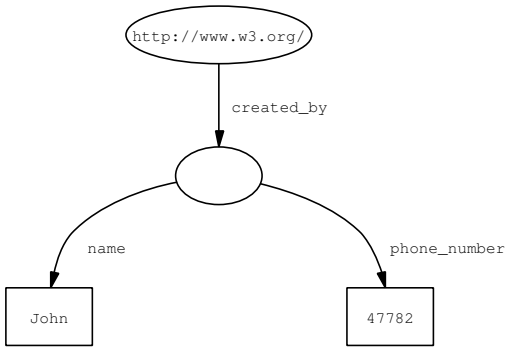


Figure 2.1: Graph representation of triple set.

These example notations reveal that RDF is ignorant about syntax; it only provides a model for representing metadata. The triple list is one possible representation, as is the labeled graph, and other syntactic representations are possible. Of course, XML would be an obvious candidate for an alternative representation. The specification of the data model includes such an XML-based encoding for RDF. In this syntax, the triples above could be expressed as follows, but other representations are possible as well.

```
<?xml version="1.0"?>
<rdf:RDF>
  <rdf:Description rdf:about="http://www.w3.org/">
    <created_by>
      <rdf:Description>
        <name>John</name>
        <phone_number>47782</phone_number>
      </rdf:Description>
    </created_by>
  </rdf:Description>
</rdf:RDF>
```

As with XML, an RDF model does not define (a priori) the semantics of any application domain or make assumptions about a particular application domain. It just provides a domain-neutral mechanism to describe metadata. Defining domain-specific properties and their semantics requires additional facilities.

2.1.4 Defining an RDF vocabulary: RDF Schema

Basically, RDF Schema is a simple type system for RDF. It provides a mechanism to define domain-specific properties and classes of resources to which you can apply those properties.

The basic modeling primitives in RDF Schema are class definitions and subclassof statements (which together allow the definition of class hierarchies), property definitions and subproperty-of statements (to build property hierarchies), domain and range statements (to restrict the possible combinations of properties and classes), and type statements (to declare a resource as an instance of a specific class). With these primitives you can build a schema for a specific domain. In the example we've been using throughout this tutorial, you could define a schema that declares two classes of resources, Person and WebPage, and two properties, name and phone, both with the domain Person and range Literal. You could use this schema to define the resource `http://www.w3.org/` as an instance of WebPage and the anonymous resource as an instance of Person. Together, this would give some interpretation and validation possibilities to the RDF data.

RDF Schema is quite simple compared to full-fledged knowledge representation languages. Also, it still does not provide exact semantics. However, this omission is partly intentional; the W3C foresees and advocates further extensions to RDF Schema.

Because the RDF Schema specification is also a kind of metadata, you can use RDF to encode it. This is exactly what occurs in the RDF Schema specification document. Moreover, the specification provides an RDF Schema document that defines the properties and classes that the RDF Schema specification introduced. As with the XML Schema specification, such a recursive definition of RDF Schema looks somewhat confusing.

2.1.5 OIL

This section offers a very brief description of the OIL language; more details can be found in (Horrocks et al., 2000). OIL, which stands for Ontology Inference Layer, is a Web-based representation and inference layer for ontologies. It unifies three important aspects provided by different communities: formal semantics and efficient reasoning support as provided by Description Logics, epistemologically rich modeling primitives as provided by the Frame community, and a standard proposal for syntactical exchange notations as provided by the Web community. A small example of an ontology in OIL is presented in figure 2.2. The language has been designed such that:

1. it provides most of the modeling primitives commonly used in frame-based and Description Logic (DL) oriented Ontologies;

2. it features simple, clean and well-defined first-order semantics;

3. automated reasoning support, (e.g., class consistency and subsumption checking) can be provided. The FaCT system (Bechhofer et al., 1999), a DL reasoner developed at the University of Manchester, can be—and has been—used to this end (Stuckenschmidt, 2000).

This core language is expected to be extended in the future with sets of additional primitives. It should be noted however, that full reasoning support may not be available for ontologies using such primitives.

```
ontology-container
    title "African Animals"
    creator "Ian Horrocks"
    subject "animal, food, vegetarians"
    description "A didactic example ontology
        describing African animals and plants"
    description.release "2.0"
    publisher "I. Horrocks"
    type "ontology"
    format "pdf"
    identifier "http://.../oil-rdfs.pdf"
    source "http://www.africa.com/"
    language "en-uk"

ontology-definitions
    slot-def eats
        inverse is-eaten-by

    slot-def has-part
        inverse is-part-of
        properties transitive

    slot-def weight
        range (min 0)
        properties functional

    slot-def color
        range string
        properties functional

    class-def animal

    class-def plant

    disjoint animal plant
    class-def tree
        subclass-of plant

    class-def branch
        slot-constraint is-part-of
            has-value tree

    class-def leaf
        slot-constraint is-part-of
            has-value branch

class-def defined carnivore
    subclass-of animal
    slot-constraint eats
        value-type animal

class-def defined herbivore
    subclass-of animal
    slot-constraint eats
        value-type (plant or
        (slot-constraint is-part-of
            has-value plant))
disjoint carnivore herbivore
class-def mammal
    subclass-of animal

class-def elephant
    subclass-of herbivore mammmal
    slot-constraint eats
        value-type plant
    slot-constraint color
        has-filler "grey"

class-def defined african-elephant
    subclass-of elephant
    slot-constraint comes-from
        has-filler Africa

class-def defined indian-elephant
    subclass-of elephant
    slot-constraint comes-from
        has-filler India

disjoint-covered elephant by
        african-elephant indian-elephant
——— instance information ———

instance-of Africa continent
instance-of Asia continent
related is-part-of India Asia
```

Figure 2.2: An example OIL ontology, modeling the animal kingdom

An ontology in OIL is represented via an *ontology container* and an *ontology definition* segment. For the container, we adopt the components defined by Dublin Core Metadata Element Set, Version 1.1[1].

The ontology-definition segment consists of an optional import statement, an optional rule base and class, slot and axiom definitions.

A class definition (**class-def**) associates a class name with a class description. This class description, in turn, consists of a subclass-of statement and zero or more slot constraints, as well as the type of the definition. (If that definition is primitive, the stated conditions for class membership are necessary, but not sufficient. If it is defined, these conditions are both necessary and sufficient).

The value of a **subclass-of** statement is a (list of) class-expression(s). This can be either a class name, a slot constraint, or a boolean combination of class expressions using the operators **and**, **or** and **not** with the standard DL semantics.

In some situations it is possible to use a *concrete-type-expression* instead of a class expression. A concrete-type-expression defines a range over some data type. Two data types that are currently supported in OIL are **integer** and **string**. Ranges can be defined using the expressions (**min** X), (**max** X), (**greater-than** X), (**less-than** X), (**equal** X) and (**range** X Y). For example, (**min** 21) defines the data type consisting of all the integers greater than or equal to 21. Another example is (**equal** "xyz"), which defines the data-type consisting of the string "xyz".

A slot constraint (**slot-constraint**) is a list of one or more constraints (restrictions) applied to a slot (property). Typical constraints are:

- **has-value (class-expr)** Every instance of the class defined by the slot constraint must be related, via the slot relation, to an instance of each class expression in the list.
- **value-type (class-expr)** *If* an instance of the class defined by the slot constraint is related via the slot relation to some individual x, *then* x must be an instance of each class expression in the list.
- **max-cardinality n (class-expr)** An instance of the class defined by the slot constraint can - at the most - be related to n distinct instances of the class expression via the slot relation (also min-cardinality and, as a shortcut for both min and max, cardinality).

A slot definition (**slot-def**) associates a slot name with a slot definition. A slot definition specifies global constraints that apply to the slot relation. A slot-def can consist of a **subslot-of** statement, **domain** and **range** restrictions, and additional qualities of the slot, such as **inverse** slot, transitive, and symmetric.

An *axiom* asserts some additional facts about the classes in the ontology, for example that the classes carnivore and herbivore are disjoint (that is, have no instances in common). Valid axioms are:

- **disjoint (class-expr)+** All of the class expressions in the list are pairwise disjoint.

[1] See http://purl.org/DC/

- **covered (class-expr) by (class-expr)+** Every instance of the first class expression is also an instance of at least one of the class expressions in the list.
- **disjoint-covered (class-expr) by (class-expr)+** Every instance of the first class expression is also an instance of exactly one of the class expressions in the list.
- **equivalent (class-expr)+** All of the class expressions in the list are equivalent (i.e. they have the same instances).

The syntax of OIL is geared towards XML and RDF. Horrocks et al. (2000) defines a DTD and a XML schema definition for OIL. In the following section (2.2), we will derive the RDFS syntax for OIL.

2.1.6 DAML+OIL and OWL

The OIL language has been succeeded by new ontology languages, namely DAML+OIL (Connolly et al., 2001) and OWL (McGuinness and van Harmelen, 2004). DAML+OIL is the outcome of a joint initiative of the DAML program[2] and the OIL language group. The resulting language inherits the formal semantics (based on a description logic) from OIL and the RDF based syntax from both. The first version of the DAML+OIL language has been published in December 2000, a second version in March 2001.

There are a few differences between OIL and DAML+OIL, which are described at the DAML+OIL. website[3]. We cite the most notable differences below.

- The RDF syntax is different. This includes simple name changes, e.g., `oil:Not` has become `daml:complementOf`, but also more excessive encoding differences, like the replacement of `oil:hasPropertyRestriction` by `daml:subClassOf`. As a consequence, the RDF encoding of DAML+OIL has lost some features of the OIL RDF encoding. The original OIL language allowed for the explicit construction of "frame-like" expressions, i.e. things with a number of superclasses and slot restrictions, as well as the ability to add axioms. In DAML+OIL, a lot of definitions are "collapsed" into axioms. This gives DAML+OIL a much more "logic" flavor instead of the a "frame" flavor of OIL (Bechhofer et al., 2001). Most notable syntax difference is the use of lists in DAML+OIL, which required an extension to standard RDF.
- OIL has better "backwards compatibility" with RDFS. In case of defined (non-primitive) concepts, half of the two way implication is still accessible to RDFS agents, because of the use of `rdfs:subClassOf`.
- DAML+OIL has an explicit `samePropertyAs` property. In OIL this should be expressed using mutual `rdfs:subPropertyOf` statements.
- DAML+OIL has two mechanisms to state disjointness. DAML+OIL provides both a `disjointWith` property that can be used to assert that two classes are disjoint and a "Disjoint" class that can be used to assert pairwise disjointness amongst all the classes in a list. OIL simply uses **disjoint** to assert disjointness amongst two or more classes.

[2] See `http://www.daml.org/`.
[3] `http://www.daml.org/2000/12/differences-oil.html`

- Different property characteristics are supported: DAML+OIL does not support `SymmetricProperty`, whereas OIL does not support `UnambiguousProperty`. However, logically both notions can be expressed via the combination of other characteristics.

OWL is the result of a standardization process of the DAML+OIL language by the W3C, the World Wide Web Consortium.[4] OWL has become a W3C Recommendation (i.e. a standard) in February 2004. The language is very similar to DAML+OIL. The RDF syntax for OWL has only minor changes from DAML+OIL. The specification documents (see McGuinness and van Harmelen, 2004) mentions the following differences between DAML+OIL and OWL:

- qualified number restrictions are not present anymore in OWL;
- the ability to directly state that properties can be symmetric is added to OWL, and
- there are a number of changes to the names of the various constructs.

Another new feature of OWL is that it has been divided in three increasingly-expressive sublanguages, OWL Lite, OWL DL, and OWL Full, targeted at different groups of users. OWL Lite provides the constructs for users that primarily need a classification hierarchy and simple constraints. OWL DL supports those users who want the maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time). OWL Full is meant for users who want maximum expressiveness and the syntactic freedom of RDF with no computational guarantees. A detailed overview of the OWL Full knowledge model and the differences between the sublanguages is given in Section 5.3.

2.1.7 Summary

XML and RDF are different formalisms with their own purposes, and their roles in the realization of the Semantic Web vision will be different. XML aims to provide an easy-to-use syntax for Web data. With it, you can encode all kinds of data that is exchanged between computers, using XML Schemas to prescribe the data structure. This makes XML a fundamental language for the Semantic Web, in the sense that many techniques will probably use XML as their underlying syntax.

XML does not provide any interpretation of the data beforehand, so it does not contribute much to the “semantic” aspect of the Semantic Web. RDF provides a standard model to describe facts about Web resources, which gives some interpretation to the data. RDF Schema extends those interpretation possibilities somewhat more. However, to realize the Semantic Web vision, it will be necessary to express even more semantics of data. For this, languages such as OIL, DAML+OIL and OWL can be used, which add new modeling primitives and formal semantics to RDF Schema.

[4] `http://www.w3c.org`

2.2 Representing Schema Languages in RDFS

In this section, we will show how RDFS can be extended to contain a more expressive knowledge representation language, which would enrich it with the required additional expressivity and the semantics of that language. We will do this by describing the ontology language OIL as an extension of RDFS. The described mechanism also is the basis of the RDF representations of DAML+OIL and OWL.

2.2.1 OIL as an extension of RDF Schema

RDF provides basic modeling primitives: ordered triples of objects and links. RDFS enriches this basic model by providing a vocabulary for RDF, which is assumed to have a certain semantics. This section presents a careful analysis of the relation between RDFS and OIL by defining OIL in RDFS, using existing vocabulary wherever possible. The reason for this is twofold. First, by re-using RDFS primitives we are effectively imposing formal semantics on them, specifically the formal semantics of OIL. Secondly, because we only extend RDFS with new primitives where necessary, RDFS becomes a full sub-language of OIL, thus providing backward compatibility from OIL to RDFS.

The complete schema can also be found at `http://www.ontoknowledge.org/oil/rdf-schema/`. The RDFS serialization of the example from the previous section is available at `http://www.ontoknowledge.org/oil/a-animals.rdfs`.

The ontology container and import mechanism

The outer box of the OIL specification in RDFS is defined by the XML prologue and the namespace definitions “xmlns:rdf” and “xmlns:rdfs”, which refer to RDF and RDFS, respectively. Namespace definitions make externally defined RDF constructs available for local use. Thus, the OIL specification uses RDF and RDFS, and an actual ontology in OIL has namespace definitions which make both the RDF and RDFS definitions as well as the OIL specification itself available.

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:oil="http://www.ontoknowledge.org/oil/rdf-schema/2000/11
                                                /10-oil-standard"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:dcq="http://purl.org/dc/qualifiers/1.1/"}
    <!-- The ontology defined in OIL with RDFS syntax-->
</rdf:RDF>
```

It is important to note that namespace definitions are not import statements and are, therefore, not transitive. An actual ontology also has to define the namespaces for RDF and RDFS via “xmlns:rdf” and “xmlns:rdfs”, otherwise, all elements of OIL that directly correspond to RDF and RDFS elements would be unavailable.

OIL’s **ontology-container** provides metadata describing an OIL ontology. Because the structure and RDF format of the Dublin Core element set is used, it is sufficient to

import the namespace of the Dublin Core element set. It is important to note that an OIL ontology's provision of a container definition is an *informal* guideline in its RDFS syntax, because it is impossible to enforce this in the schema definition.

Aside from the container, an OIL ontology consists of a set of definitions. The **import** definition is a simple list of references to other OIL modules to be included in the ontology. We make use of the XML namespace mechanism to incorporate this mechanism in our RDFS specification. Again, in contrast to the import statement in OIL, "inclusion" via the namespace definition is not transitive.

Class and attribute definitions

In OIL, a class definition links a class with a name, documentation, a type, its superclasses, and the attributes defined for it. In RDFS, classes are simply declared by assigning them a name (with the ID attribute). We will demonstrate how to write OIL class definitions in RDF, making maximum use of existing RDFS constructs, but extending RDFS with additional constructs where necessary (see table 2.1 and figure 2.3). We have followed the informal RDF guideline of starting property names with a lower-case letter, and class names with a capital.

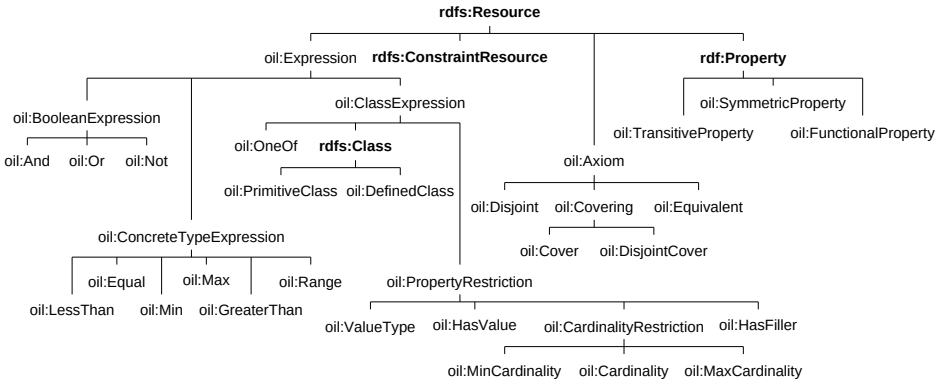


Figure 2.3: The OIL extensions to RDFS in the subsumption hierarchy.

To illustrate the use of these extensions, we will examine them systematically with various example OIL class definitions that need to be represented in RDFS syntax:

class-def defined herbivore
  subclass-of animal
    slot-constraint *eats*
      value-type (plant **or**
      (**slot-constraint** *is-part-of* **has-value** plant))

class-def elephant
  subclass-of herbivore mammal

slot-constraint *eats*
  value-type plant
slot-constraint *color*
  has-filler “grey”

The first defines a class ”herbivore”, a subclass of animal, whose instances eat plants or parts of plants. The second defines a class ”elephant”, which is a subclass of both herbivore and mammal.

Defined classes and Primitive classes We start by translating the first class definition. The header can be done in a straightforward manner, using the rdfs:Class construct and the rdf:ID property to assign a name:

```
<rdfs:Class rdf:ID="herbivore"> </rdfs:Class>
```

This definition does not yet clearly identify this class as a defined class. We chose to introduce two extra classes in the OIL namespace: PrimitiveClass and DefinedClass. In a particular class definition, we can use one of these two options to identify a class as a defined class:

```
<rdfs:Class rdf:ID="herbivore">
  <rdf:type rdf:resource="http://www.ontoknowledge.org/oil/
              rdf-schema/2000/11/10-oil-standard#DefinedClass"/>
</rdfs:Class>
```

or:

```
<oil:DefinedClass rdf:ID="herbivore"> </oil:DefinedClass>
```

We will use the first method of serialization throughout this chapter, but it is important to note that both models are exactly the same.

This method of making an actual class an instance of either DefinedClass or PrimitiveClass introduces a nice object-meta distinction between the OIL RDFS schema and the actual ontology: using rdf:type, we can consider the class “herbivore” to be an *instance* of DefinedClass. Generally speaking, if a class in OIL is not explicitly identified as a defined class, it is assumed to be primitive.

Class Subsumption Next, we need to translate the subclass-of statement to RDFS. This also can be done in a straightforward manner, simply by re-using existing RDFS expressiveness:

```
<rdfs:Class rdf:ID="herbivore">
  <rdf:type rdf:resource="http://www.ontoknowledge.org/oil/
     rdf-schema/2000/11/10-oil-standard#DefinedClass"/>
  <rdfs:subClassOf rdf:resource="#animal"/>
</rdfs:Class>
```

However, in order to define a class as a subclass of a class *expression*, we would need to use the oil:subClassOf property.

Slot Constraints We still need to serialize the slot constraint on the class "herbivore". RDFS provides no mechanism for restricting the attributes of a class on a local level. This is due to the property-centric nature of the RDF data model: properties are defined globally, with their domain description coupling them to the relevant classes.

To overcome this problem, we will introduce the oil:hasPropertyRestriction property, which is an rdf:type of rdfs:ConstraintProperty (analogous to rdfs:domain and rdfs:range). In doing so, we will be using RDFS's full potential capacity for extensibility. We will also introduce oil:PropertyRestriction as a placeholder class[5] for specific classes of slot constraints, such as has-value, value-type, cardinality, etc. These are all modeled in the OIL namespace as subclasses of oil:PropertyRestriction:

```
<rdfs:Class rdf:ID="ValueType">
  <rdfs:subClassOf rdf:resource="#PropertyRestriction"/>
</rdfs:Class>
```

They are also similar for the other slot constraints. For the three cardinality constraints, an extra property "number" will be introduced, which will serve to assign a concrete value to the cardinality constraints.

To connect a ValueType slot constraint with its actual values, such as the property to which it refers and the class to which it restricts that property, we will introduce a pair of helper properties. These helper properties have no direct counterpart in terms of OIL primitives, but do serve to connect two classes. We will define a property oil:onProperty to connect a property restriction with the subject property, and a property oil:toClass to connect the property restriction to its class restriction.

In our example ontology, we would serialize the first part of the slot constraint using the primitives introduced above. This would proceed as follows:

```
<rdfs:Class rdf:ID="herbivore">
  <rdf:type rdf:resource="http://www.ontoknowledge.org/oil/
     rdf-schema/2000/11/10-oil-standard#DefinedClass"/>
  <rdfs:subClassOf rdf:resource="#animal"/>
  <oil:hasPropertyRestriction>
    <oil:ValueType>
      <oil:onProperty rdf:resource="#eats"/>
      <oil:toClass> </oil:toClass>
    </oil:ValueType>
  </oil:hasPropertyRestriction>
</rdfs:Class>
```

To restrict the value type of a property to a string or an integer, we could use the toConcreteType property:

```
...
  <oil:ValueType>
    <oil:onProperty rdf:resource="#age"/>
    <oil:toConcreteType rdf:resource="http://www.ontoknowledge.org/
           oil/rdf-schema/2000/11/10-oil-standard#Integer"/>
  </oil:ValueType>
...
```

[5] A placeholder class in the OIL RDFS specification is only used to apply domain- and range restrictions to a group of classes, and will not be used in the actual OIL ontology.

Boolean Expressions The slot constraint has not been completely translated yet: the toClass element is not yet filled. Here we come across a feature of OIL that is not available in RDFS: the *boolean expression*. A boolean expression in OIL is an expression that evaluates to either a class definition or a concrete type. In the case of a class definition, such an expression is a boolean combination of classes and/or slot constraints. In the case of a concrete type definition, the expression can be a simple string or integer value, or a more complex expression (see Section 8). In the example, we have a boolean 'or' expression that evaluates to the class of all things that are plants or parts of plant.

We will introduce oil:Expression as a common placeholder, along with oil:ConcreteTypeExpression and oil:ClassExpression as specialization placeholders. However, oil:BooleanExpression will be introduced as a sibling of these two, since we want to be able to construct boolean expressions with either kind of expression. The specific boolean operators, 'and', 'or' and 'not', are introduced as subclasses. We should also note that since a single class is essentially a simple kind of class expression, rdfs:Class itself should be a subclass of oil:ClassExpression (see figure 2.3).

The 'and', 'or' and 'not' operators are connected to operands using the oil:hasOperand property. Again, this property has no direct equivalent in OIL primitive terms. Rather, it serves as a helper in connecting two class expressions, as the only way to relate two classes in the RDF data model is by means of a Property.

In our example, we need to serialize a boolean 'or'. The RDF Schema definition of the operator reads as follows:

```
<rdfs:Class rdf:ID="Or">
  <rdfs:subClassOf rdf:resource="#BooleanExpression"/>
</rdfs:Class>
```

The helper property is defined as follows:

```
<rdf:Property rdf:ID="hasOperand">
  <rdfs:domain rdf:resource="#BooleanExpression"/>
  <rdfs:range rdf:resource="#ClassExpression"/>
</rdf:Property>
```

The fact that hasOperand is only to be used on boolean class expressions is expressed using the rdfs:domain construction. This type of modeling stems directly from the RDF property-centric approach.
Now, we apply what we defined above to the example:

```
<rdfs:Class rdf:ID="herbivore">
  <rdf:type rdf:resource="http://www.ontoknowledge.org/oil/
     rdf-schema/2000/11/10-oil-standard#DefinedClass"/>
  <rdfs:subClassOf rdf:resource="#animal"/>
  <oil:hasPropertyRestriction>
    <oil:ValueType>
      <oil:onProperty rdf:resource="#eats"/>
      <oil:toClass>
        <oil:Or>
          <oil:hasOperand rdf:resource="#plant"/>
          <oil:hasOperand>
            <HasValue>
              <oil:onProperty rdf:resource="#is-part-of"/>
              <oil:toClass rdf:resource="#plant"/>
```

```
            </HasValue>
          </oil:hasOperand>
        </oil:Or>
      </oil:toClass>
    </oil:ValueType>
  </oil:hasPropertyRestriction>
</rdfs:Class>
```

Observe that the HasValue property restriction is not related to the class by a hasPropertyRestriction property, but by a hasOperand property. This stems from the fact that the property restriction plays the role of a boolean operand here.

Lists of statements Now, we will illustrate some more features by translating the second class definition, "elephant". The first bit is trivial:

```
<rdfs:Class rdf:ID="elephant"> </rdfs:Class>
```

Next, we need to translate the OIL subsumption statement to RDFS. This statement contains a list of superclasses. In the RDFS syntax, we will model these as seperate subClassOf statements:

```
<rdfs:Class rdf:ID="elephant">
  <rdfs:subClassOf rdf:resource="#mammal"/>
  <rdfs:subClassOf rdf:resource="#herbivore"/>
</rdfs:Class>
```

Next, we have two slot constraints. The first of these is a value-type restriction, and is serialized in the same manner demonstrated in the "herbivore" example:

```
<rdfs:Class rdf:ID="elephant">
  <rdfs:subClassOf rdf:resource="#mammal"/>
  <rdfs:subClassOf rdf:resource="#herbivore"/>
  <oil:hasPropertyRestriction>
    <oil:ValueType>
      <oil:onProperty rdf:resource="#eats"/>
      <oil:toClass rdf:resource="#plant"/>
    </oil:ValueType>
  </oil:hasPropertyRestriction>
</rdfs:Class>
```

Slot constraints to concrete types The second slot constraint is a restriction to a particular concrete type. In OIL, a shortcut syntax for such restrictions has been introduced in the form of a "has-filler" primitive. We will serialize this as we did with the other slot constraints: we will introduce a class oil:HasFiller and helper properties, oil:stringFiller and oil:integerFiller, to connect to the value:

```
<oil:HasFiller>
  <oil:onProperty rdf:resource="#color"/>
  <oil:stringFiller>grey</oil:stringFiller>
</oil:HasFiller>
```

Unfortunately, there is no direct way in RDFS to constrain the value of a property to a particular datatype. Therefore, the range value of oil:stringFiller can not be constrained

to contain only strings. We will create two subclasses of rdfs:Literal, named oil:String and oil:Integer only for the sake of clarity.

```
<rdfs:Class rdf:ID="String">
   <rdfs:comment>
   The subset of Literals that are strings.
   </rdfs:comment>
   <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/
                                          rdf-schema#Literal"/>
</rdfs:Class>
```

The range of the filler properties can now be set to the appropriate class, although it is still possible to use any type of Literal. The semantics of rdfs:Literal are only that anything of this type is atomic, i.e. it will not be processed further by an RDF processor. The fact that in this case it should be a string value can only be made an informal guideline.

```
<rdf:Property ID="stringFiller">
  <rdfs:domain rdf:resource="#HasFiller"/>
  <rdfs:range rdf:resource="#String"/>
</rdf:Property>
```

Using all this, we get the following complete translation of the class “elephant”:

```
<rdfs:Class rdf:ID="elephant">
  <rdfs:subClassOf rdf:resource="#mammal"/>
  <rdfs:subClassOf rdf:resource="#herbivore"/>
  <oil:hasPropertyRestriction>
    <oil:ValueType>
      <oil:onProperty rdf:resource="#eats"/>
      <oil:toClass rdf:resource="#plant"/>
    </oil:ValueType>
    <oil:HasFiller>
      <oil:onProperty rdf:resource="#color"/>
      <oil:stringFiller>grey</oil:stringFiller>
    </oil:HasFiller>
  </oil:hasPropertyRestriction>
</rdfs:Class>
```

Note that more than one property restriction is allowed within the hasPropertyRestriction element.

Conclusion The serialization we propose gives us enough expressiveness to translate any possible OIL class definition to an RDF syntax. Use of RDF(S) specific constructs is maximized without sacrificing clarity of the specification. This is to enable RDF agents that are not OIL-aware to understand as much of the specification as possible, while retaining the option of translating back to OIL unambiguously. In the next section, we will examine how to serialize global slot definitions.

Slot definitions

Both OIL and RDFS allow slots as first-class citizens of an ontology. Therefore, slot definitions in OIL map nicely onto property definitions in RDFS. In addition, the “subslot-of”, “domain”, and “range” properties have almost direct equivalents in RDFS. Table 2.2 presents an overview of the OIL constructs and the corresponding RDFS constructs.

OIL primitive	RDFS syntax	type
class-def	`rdfs:Class`	class
subclass-of	`rdfs:subClassOf`	property
class-expression	*`oil:ClassExpression`* (placeholder only)	class
and	`oil:And` (subclass of BooleanExpression)	class
or	`oil:Or` (subclass of BooleanExpression)	class
not	`oil:Not` (subclass of BooleanExpression)	class
slot-constraint	*`oil:PropertyRestriction`* (placeholder only) `oil:hasPropertyRestriction` (rdf:type of rdfs:ConstraintProperty) *`oil:CardinalityRestriction`* (placeholder only) (subclass of oil:PropertyRestriction)	class property class
has-value	`oil:HasValue` (subclass of oil:PropertyRestriction)	class
has-filler	`oil:HasFiller` (subclass of oil:PropertyRestriction)	class
value-type	`oil:ValueType` (subclass of oil:PropertyRestriction)	class
max-cardinality	`oil:MaxCardinality` (subclass of oil:CardinalityRestriction)	class
min-cardinality	`oil:MinCardinality` (subclass of oil:CardinalityRestriction)	class
cardinality	`oil:Cardinality` (subclass of oil:CardinalityRestriction)	class

Table 2.1: Class-definitions in OIL and the corresponding RDF(S) constructs

There are a few subtle differences between domain and range restrictions in OIL and their equivalents in RDFS. OIL allows multiple domain and range restrictions on a single slot. The interpretation of such a set of restrictions is the *intersection* of the classes in the individual statements (conjunctive semantics). In RDFS, multiple domain statements are allowed, but their interpretation is the *union* of the classes in the statements (disjunctive semantics). This limits the reasoning capabilities of RDFS drastically[6].

Despite these semantics for domain, a Property can have at most one range restriction in RDFS. However, the current consensus within the RDF community is that the semantics of domain and range should change in the next release of RDFS. We anticipate such a change, and will interpret both multiple domain and multiple range restrictions with conjunctive semantics.

Another difference with RDFS is that OIL not only allows classes as range and domain of properties, but also class *expressions*, and – in the case of range – concrete-type

[6]For example, it is never possible to derive class membership from a domain statement when union semantics are used.

expressions. It is not possible to reuse rdfs:range and rdfs:domain for these sophisticated expressions, because of the conjunctive semantics of multiple range statements: we cannot extend the range of rdfs:range or rdfs:domain, we can only restrict it. In our RDFS serialization of OIL, we will, therefore, introduce two new ConstraintProperties oil:domain and oil:range. They have the same domain as their RDFS equivalent (i.e., rdf:Property), but have a broader range. For domain, class expressions are valid fillers; for range, both class expressions and concrete type expressions may be used:

```
<rdfs:ConstraintProperty rdf:ID="domain">
  <rdfs:domain rdf:resource="http://www.w3.org/1999/02/
                             22-rdf-syntax-ns#Property"/>
  <rdfs:range rdf:resource="#ClassExpression"/>
</rdfs:ConstraintProperty>

<rdfs:ConstraintProperty rdf:ID="range">
  <rdfs:domain rdf:resource="http://www.w3.org/1999/02/
                             22-rdf-syntax-ns#Property"/>
  <rdfs:range rdf:resource="#Expression"/>
</rdfs:ConstraintProperty>
```

When translating a slot definition, rdfs:domain and rdfs:range should be used for simple (one class) domain and range restrictions. For example:

slot-def *gnaws*
  subslot-of *eats*
  domain Rodent

will be translated into:

```
<rdf:Property rdf:ID="gnaws">
  <rdfs:subPropertyOf rdf:resource="#eats"/>
  <rdfs:domain rdf:resource="#Rodent"/>
</rdf:Property>
```

For more complicated statements, the oil:range or oil:domain properties should be used:

slot-def *age*
  domain (elephant **or** lion)
  range (**range** 0 70)

is in the RDFS representation:

```
<rdf:Property rdf:ID="age">
  <oil:domain>
    <oil:Or>
      <oil:hasOperand rdf:resource="#elephant"/>
      <oil:hasOperand rdf:resource="#lion"/>
    </oil:Or>
  </oil:domain>
  <oil:range>
    <oil:Range>
      <oil:integerValue>0</oil:integerValue>
      <oil:integerValue>70</oil:integerValue>
    </oil:Range>
  </oil:range>
</rdf:Property>
```

To specify that the range of a property is string or integer, we will use our definitions of oil:String and oil:Integer as subclasses of rdfs:Literal. For example, in stating that the range of age is integer, we could say:

```
<rdf:Property ID="age">
  <rdfs:range rdf:resource="http://www.ontoknowledge.org/oil/
     rdf-schema/2000/11/10-oil-standard#Integer">
</rdf:Property>
```

However, global slot-definitions in OIL allow specification of more aspects of a slot than do property definitions in RDFS. Aside from the domain and range restrictions, OIL slots can also have an "inverse" attribute and qualities like "transitive" and "symmetric".

In light of this, we will add a property "inverseRelationOf" with "rdf:Property" as the domain and range. We also add the classes "TransitiveProperty", "FunctionalProperty" and "SymmetricProperty" to reflect the different qualities of a slot. In the RDFS serialization of OIL, the rdf:type property can be used to add a quality to a property. For example, the OIL definition of:

slot-def *has-part*
  inverse *is-part-of*
  properties transitive

reads as follows in RDFS:

```
<rdf:Property rdf:ID="has-part">
  <rdf:type rdf:resource="http://www.ontoknowledge.org/oil/
     rdf-schema/2000/11/10-oil-standard#TransitiveProperty"/>
  <oil:inverseRelationOf rdf:resource="#is-part-of"/>
</rdf:Property>
```

In the abbreviated syntax, it reads:

```
<oil:TransitiveProperty rdf:ID="has-part">
  <oil:inverseRelationOf rdf:resource="#is-part-of"/>
</oil:TransitiveProperty>
```

This method of translating the qualities of properties features the same nice object-meta distinction (between the OIL language and the actual ontology) as the translation of the "type" of a class (see section 2.2.1). In an actual ontology, the property "has-part" can be considered as an *instance* of a TransitiveProperty. A property can be made an instance of more than one class, and thus assigned multiple qualities. Note that this way of representing qualities of properties in RDFS follows the proposed general approach of modeling axioms in RDFS, presented in (Staab et al., 2000). This approach makes the same distinction between language-level constructs and schema-level constructs.

One alternative way of serializing the attributes of properties would be to define the qualities "transitive" and "symmetric" as subproperties of rdf:Property. Properties in the actual ontology (e.g. "has-part") would, in turn, be defined as subProperties of these qualities (e.g. transitiveProperty). However, this would mix-up the use of properties at the OIL specification level as well as at the actual ontology level.

A third approach would be to model the qualities again as subproperties of rdf:Property, but to define properties in the actual ontology as instances (rdf:type) of such qualities.

This approach preserves the object-meta level distinction. However, we dislike the use of rdfs:subPropertyOf at the meta level, because then rdfs:subPropertyOf has two meanings, at the meta level and at the object level.

In our opinion, the first solution is preferable, because of the clean distinction it makes between the meta and object level.

OIL primitive	RDFS syntax	type
slot-def	`rdf:Property`	class
subslot-of	`rdfs:subPropertyOf`	property
domain	`rdfs:domain`	property
	`oil:domain`	property
range	`rdfs:range`	property
	`oil:range`	property
inverse	`oil:inverseRelationOf`	property
transitive	`oil:TransitiveProperty`	class
functional	`oil:FunctionalProperty`	class
symmetric	`oil:SymmetricProperty`	class

Table 2.2: Slot-definitions in OIL and the corresponding RDF(S) constructs.

Axioms

Axioms in OIL are factual statements about the classes in the ontology. They correspond to n-ary relations between class expressions, where n is 2 or greater.

RDF features only binary relations (properties). Therefore, we cannot simply map OIL axioms to RDF properties. Instead, we chose to model axioms as classes, with helper properties connecting them to the class expressions involved in the relation. Since axioms can be considered objects, this is a very natural approach towards modeling them in RDF (see also (Staab and Mädche, 2000; Staab et al., 2000)). Note also that binary relations (properties) are modeled as objects in RDFS as well (i.e., any property is an instance of the class rdf:Property). We simply introduce a new primitive *alongside* rdf:Property for relations with higher arity (see figure 2.3).

We introduce a placeholder class oil:Axiom, and model specific types of axioms as subclasses:

```
<rdfs:Class ID="Disjoint">
  <rdfs:subClassOf rdf:resource="#Axiom"/>
</rdfs:Class>
```

We do the same for Equivalent.

We also introduce a property to connect the axiom object with the class expressions it relates to each other: oil:hasObject is a property connecting an axiom with an object class expression. We will illustrate this below by serializing the axiom that herbivores, omnivores and carnivores are (pairwise) disjoint:

```
<oil:Disjoint>
  <oil:hasObject rdf:resource="#herbivore"/>
```

```
  <oil:hasObject rdf:resource="#carnivore"/>
  <oil:hasObject rdf:resource="#omnivore"/>
</oil:Disjoint>
```

Since in a disjointness axiom (or an equivalence axiom) the relation between class expressions is bidirectional, we can connect all class expressions to the axiom object using the same type of property.

However, in a covering axiom (such as cover or disjoint-cover), the relation between class expressions is not bidirectional: one class expression may serve as the covering, while several others function as part of that covering.

For modeling covering axioms, we will introduce a separate placeholder class, oil:Covering, which is a subclass of oil:Axiom. The specific types of coverings available are modeled as subclasses of oil:Covering again:

```
<rdfs:Class ID="Cover">
  <rdfs:subClassOf rdf:resource="#Covering"/>
</rdfs:Class>

<rdfs:Class ID="DisjointCover">
  <rdfs:subClassOf rdf:resource="#Covering"/>
</rdfs:Class>
```

We will also introduce two additional properties: oil:hasSubject, to connect a covering axiom with its subject, and oil:isCoveredBy (a subproperty of oil:hasObject) to connect a covering axiom with the classes that cover the subject.

We will illustrate this below by serializing the axiom that the class animal is covered by carnivore, herbivore, omnivore, and mammal (i.e. every instance of animal is also an instance of at least one of the other classes).

```
<oil:Cover>
  <oil:hasSubject rdf:resource="#animal"/>
  <oil:isCoveredBy rdf:resource="#carnivore"/>
  <oil:isCoveredBy rdf:resource="#herbivore"/>
  <oil:isCoveredBy rdf:resource="#omnivore"/>
  <oil:isCoveredBy rdf:resource="#mammal"/>
</oil:Cover>
```

Restrictions to valid expressions

In the previous sections, we demonstrated how the knowledge representation constructs in OIL can be defined as an extension to RDF Schema. With these constructs, every OIL ontology can be fully expressed in an RDF Schema representation. However, it was not possible to define the extension in such a way that all schemas that follow it are also valid OIL ontologies. In other words, there are some restrictions to valid ontologies that are not expressible in the RDF Schema extension. [7]

First, there is a problem with data types. It cannot be enforced that instances of oil:String are really strings or that instances of oil:Integer are really integers. Consequently, it is syntactically possible to state:

[7] By "valid" we mean: not allowed by the BNF grammar of OIL. From the logical point of view, there is nothing wrong with a statement such as (dog **and** (**min** 0)); it just happens to be equivalent to the empty class.

```
<rdf:Property rdf:ID="weight">
  <rdf:range>
    <oil:Min>
      <oil:integerValue>nonsense</oil:integerValue>
    </oil:Min>
  </rdf:range>
</rdf:Property>
```

This is due to the fact that the RDF Schema specification has (intentionally) not specified any primitive data types. According to the specification, the work on data typing in XML itself should be the foundation for such a capability.

Second, the RDF Schema specification of OIL does not prevent the intertwining of boolean expressions of classes with boolean expressions of concrete data types. Although a statement like (dog **and** (**min** 0)) is not allowed in OIL, it is syntactically possible to state:

```
<oil:And>
  <oil:hasOperand rdf:resource="#Dog">
  <oil:hasOperand>
    <oil:Min>
      <oil:integerValue>0</oil:integerValue>
    </oil:Min>
  </oil:hasOperand>
</oil:And>
```

To prevent this kind of mixing, we could have introduced separate boolean operators for class expressions and concrete type expressions. In our opinion, however, this would have made the schema too convoluted.

Finally, another kind of problem is that the schema cannot prevent the unnecessary use of the OIL variants of standard RDF Schema constructs, such as oil:subClassOf, oil:range and oil:domain. Although this unnecessary use does not affect the semantics of the ontology, it limits the compatibility of ontologies with plain RDF Schema.

2.2.2 Compatibility with RDF Schema

In this section, we will discuss the extent of the compatibility that we have achieved between the semantic extension (OIL), and the underlying language (RDF Schema).

We can distinguish three levels in all ontology languages. The first of these is the ontology language itself, such as OIL. This is the language in which to state class-definitions, subclass-relations, attribute-definitions etc. The second level consists of the ontological classes (e.g. “giraffe” or “herbivore”), their subclass relations, and their properties (e.g. “eats”). Naturally, these are expressed in the language of the first level. The third level contains the instances of the ontology, such as individual giraffes or lions that belong to classes defined at the second level.

A look at the existing W3C RDF/RDF Schema recommendation would reveal the following about these levels:

1. The ontology language is, of course, RDF Schema;
2. Specific classes, their properties and relations are, therefore, written in RDF Schema, e.g.:

```
<rdfs:Class rdf:ID="herbivore">
  <rdfs:subClassOf rdf:resource="#animal">
</rdfs:Class> <rdf:Property rdf:ID="eats"/>
```

3. Instances are written in RDF (note: *not* RDF Schema), e.g.:

```
<rdf:Description about="http://www.cs.vu.nl/~frankh">
  <rdf:type rdf:resource="#herbivore"/>
</rdf:Description>
```

If we were to examine a semantic extension of RDF Schema such as OIL, we would find the following:

1. The ontology language is OIL, but it is important to realize that OIL includes RDF Schema as a sublanguage.

2. As a result, class expressions written in OIL are actually also legal RDF Schema. For example, besides being a meaningful OIL definition, the class definition of "herbivore" in item 2 above is also a legal example of an RDF Schema definition. Of course, since OIL is an *extension* of RDF Schema, not all parts of an OIL definition are *meaningful* RDF Schema. This is illustrated below.

```
<rdfs:Class rdf:ID="herbivore">
  <rdfs:subClassOf rdf:resource="#animal"/>
  <oil:hasPropertyRestriction>
    <oil:ValueType>
      <oil:onProperty rdf:resource="#eats"/>
      <oil:toClass>
        <oil:Or>
          <oil:hasOperand rdf:resource="#plant"/>
          <oil:hasOperand>
            <oil:HasValue>
              <oil:onProperty
                rdf:resource="#is-part-of"/>
              <oil:toClass rdf:resource="#plant"/>
            </oil:HasValue>
          </oil:hasOperand>
        </oil:Or>
      </oil:toClass>
    </oil:ValueType>
  </oil:hasPropertyRestriction>
</rdfs:Class>
```

   Note that the semantics of the hasPropertyRestriction statement would be impossible for an RDF Schema processor to interpret. The entire state is legal RDF syntax, so it can be parsed, but the intended semantics of the property restriction itself can only be understood by an OIL-aware application. Notice that the first subClassOf statement is still fully interpretable even by an OIL-unaware RDF Schema processor.

3. OIL instances are written as RDF! This is an important consequence of the fact that the second level is organized as an extension of RDF Schema.

The above shows that we have now achieved two important compatibility results: first, OIL is *backwardly compatible* with RDF Schema, i.e. every RDF Schema specification is also a valid OIL ontology declaration. Secondly, we have achieved *partial forward compatibility*. This means that even if an ontology is written in the richer modeling language (OIL), a processor for the simpler ontology language (RDF Schema) can still:

a) fully interpret all the instance information of the ontology, and

b) partially interpret the class-structure of the ontology. This can be achieved by simply ignoring any statement not from the rdf or rdfs namespaces. (In our example, these came from the oil namespace). In the above definition of "herbivore", for instance, an RDF Schema processor would interpret the statement simply as asserting that herbivores are a subclass of animals, and that they have some other property that it cannot interpret. This is a correct, albeit partial, interpretation of the definition.

Such partial interpretability of semantically rich meta-data by semantically poor processing agents is a crucial step towards sharing meta-data on the Semantic Web. We cannot realistically hope that all of the Semantic Web will be built on a single standard for semantically rich meta-data. The above shows that multiple semantic modeling languages do not have to lead to meta-data that are completely impossible for others to interpret. Instead, simpler processors can still pick up as much of the meta-data from rich processors as they can "understand". They can safely ignore the rest in the knowledge that their partial interpretation is still correct with respect to the original intention of the meta-data.

2.2.3 Summary

In this section, we have shown that RDFS is only a small step towards the expressiveness required for the Semantic Web. We then illustrated how RDFS can still be used to represent more expressive languages, by extending it with additional modeling primitives as defined by a more formal knowledge representation scheme, such as OIL.

An important advantage to our approach is that it maximizes the compatibility with RDFS. Not only is every RDF Schema document a valid OIL ontology declaration, every OIL ontology can also be partially interpreted by a semantically poorer processing agent. Needless to say, this partial interpretation is incomplete. All the same, it is correct under the intended semantics of the ontology. We firmly believe that our extension method is generally applicable across knowledge representation formalisms.

Chapter 3

Ontology Change Management: Problems and Solutions

***Note**: Large parts of this chapter are based on earlier publications. Section 3.1 is published in the proceedings of the IJCAI workshop on Ontologies (Klein, 2001a). Section 3.2 is accepted for publication as part of an article in KAIS (Noy and Klein, 2004).*

In the motivating use case that we described in Chapter 1, the "Semantic Web", ontologies are used to describe the meaning of data. In the previous chapter, we discussed the languages that are used to express both ontologies and data on the web.

In this chapter, we study the problem of ontology change. We do this in two ways. First, we look at the problem of ontology diversity: in Section 3.1, we describe in what aspects two ontologies can be different. Then, we look at existing strategies to handle changes in schema-like structures. In Section 3.2, we look at database schema versioning. We discuss the main issues and compare the field with ontology versioning. In Section 3.3, we describe a study of current change management strategies for large ontologies. Based on this, we list a number of ontology evolution tasks in the last section of the chapter.

3.1 Ontology Mismatches

Several problems arise when one tries to use independently developed ontologies together, or when existing ontologies are adapted for new purposes. Based on a literature study, we distinguish several types of mismatches that can occur between different ontologies.

Mismatches between ontologies are the key type of problems that hinder the combined use of independently developed ontologies. We will now explore *how* ontologies may differ. In the literature, there are a lot of possible mismatches mentioned, which are

not always easy comparable. To make them more comparable, we classify the different types of mismatches and relate them to each other.

As a first step, we will distinguish between two levels at which mismatches may appear. The first level is the **language** or meta-model level. This is the level of the language primitives that are used to specify an ontology. Mismatches at this level are mismatches between the *mechanism* to define classes, relations and so on. The second level is the **ontology** or model level, at which the actual ontology of a domain lives. A mismatch at this level is a difference in the way the domain is modeled. The distinction between these two levels of differences is made very often. Kitakami et al. (1996) and Visser et al. (1997) call these kinds of differences respectively *non-semantic* and *semantic* differences. Others make this distinction implicitly, by only concentrating on one of the two levels. For example, Wiederhold (1994) analyses domain differences (i.e., ontology level), while Grosso et al. (1998) and Bowers and Delcambre (2000) look at language level differences. In the following, we will avoid the use of the words "semantic differences" for ontology level differences, because we reserve those words for a more specific type of difference (which will be described below).

Below, we will give an overview and characterization of different types of mismatches that can appear at each of those two levels.

3.1.1 Language level mismatches

Mismatches at the language level occur when ontologies written in different ontology languages are combined. Chalupsky (2000) defines mismatches in *syntax* and *expressivity*. In total, we distinguish four types of mismatches that can occur, although they often coincide.

- **Syntax** Different ontology languages typically use different syntaxes. For example, to define a class in RDF Schema (Brickley and Guha, 2000), one uses `<rdfs:Class rdf:ID="CLASSNAME"/>`. In LOOM, the expression `(defconcept CLASSNAME)` is used to define a class. This difference is probably the simplest kind of mismatch. However, this mismatch often doesn't come alone, but is coupled with other differences at the language level. A typical example of a "syntax only" mismatch is an ontology language that has several syntactical representations. In this simple case, a rewrite mechanism is sufficient to repair these problems.
- **Logical representation** A slightly more complicated mismatche at this level is the difference in representation of logical notions. For example, in some languages it is possible to state explicitly that two classes are disjoint (e.g., `disjoint A B`), whereas in other languages it is necessary to use negation in subclass statements (e.g., `A subclass-of (NOT B), B subclass-of (NOT A)`) to express the same notion. The point here is not whether something can be expressed—the statements are logically equivalent—but which language constructs should be used to express a given notion. This mismatch is not about the representation of *concepts*, but about the representation of *logical notions*. This type of mismatch is still relatively easily solvable, e.g. by giving translation rules from one logical representation to another.

- **Semantics of primitives** A more subtle possible difference at the metamodel level is the semantics of language constructs. Despite the fact that sometimes the same name is used for a language construct in two languages, the semantics may differ; e.g., there are several interpretations of `A equalTo B`.
  Note that even when two ontologies seem to use the same syntax, the semantics can differ. For example, the OIL RDF Schema syntax (Broekstra et al., 2001) interprets multiple `<rdfs:domain>` statements as the intersection of the arguments, whereas a previous version of RDF Schema itself used union semantics.
- **Language expressivity** A final type of mismatch at the metamodel level is a difference in expressivity between two languages. This difference implies that some languages are able to express things that are not expressible in other languages. For example, some languages have constructs to express negation, others have not. Other typical differences in expressivity are the support of lists, sets, default values, etc.
  This type of mismatch has probably the largest impact. The "fundamental differences" between knowledge models that are described by Grosso et al. (1998) are close to our interpretation.

Our list of differences at the language level can be seen as more or less compatible with the broad term "language heterogeneity" of Visser et al. (1997).

3.1.2 Ontology level mismatches

Mismatches at the ontology—or model—level happen when two or more ontologies that describe (partly) overlapping domains are combined. These mismatches may occur when the ontologies are written in the same language, as well as when they use different languages. Based on the literature and on our own observations, we can distinguish several types of mismatches at the model level.

Visser et al. (1997) make a useful distinction between mismatches in the *conceptualization* and *explication* of ontologies. A conceptualization mismatch is a difference in the way a domain is interpreted (conceptualized), which results in different ontological concepts or different relations between those concepts. An explication mismatch, on the other hand, is a difference in the way the conceptualization is *specified*. This can manifest itself in mismatches in definitions, mismatches in terms and combinations of both. Visser et al. list all the combinations. Four of these combinations are related to homonym terms and synonym terms.

Wiederhold (1994) also mentions problems with synonym terms (called *naming differences*) and homonym terms (*subjective meaning*). Besides that, he describes possible differences in the *scope of concepts*, which is an example of a conceptual mismatch. Finally, he mentions *value encoding* differences, for example, differences in the unit of measurement.

Chalupsky (2000) lists four types of mismatches in ontologies. One of these, namely the *inference system bias* is in our opinion not a real mismatch. A inference system bias means that a different modeling style is chosen because of a specific reasoning task. This probably results in mismatches, but is not a mismatch in itself. The other three

mismatches, *modeling conventions*, *coverage and granularity* and *paradigms* can be categorized as instances of the two main mismatch types of Visser et al.. We will describe these mismatches below.

We now relate the different types of mismatches that are distinguished by the authors cited above. The first two mismatches at the model level that we distinguish are instances of the **conceptualization mismatches** of Visser et al.. These are semantic differences, i.e., not only the specification, but also the conceptualization of the domain (see the definition of Gruber, 1993) is different in the ontologies that are involved. The detection and reconciliation of conceptualization differences usually requires the knowledge of a domain expert.

- **Concept scope** Two classes seem to represent the same concept, but do not have exactly the same instances, although these intersect. The standard example is the class "employee": several administrations use slightly different concepts of employee, as mentioned by Wiederhold (1994). Visser et al. (1997) call this a *class mismatch* and specifies it further into specific types.
- **Model coverage** This is a mismatch in the things that are contained in the ontology. There are three dimensions for model coverage. A first dimension is the *extent* of the model, i.e. the things at the periphery of the domain that are included or not included. A second dimension is the *granularity* of the model, i.e. the level of detail in which a domain is described. Finally, there is the *perspective* of the ontology which determines what aspects of a domain are described (Borst, 1997). Models can be different in each of these dimensions. For example, an ontology about public transport might or might not include taxis (difference in extent), might distinguish many different types of trains or not (difference in granularity), and could describe technical aspects or functional aspects (difference in perspective).

The other ontology-level mismatches can be categorized as **explication mismatches**, in the terminology of Visser et al.. The first two of these result from explicit choices of the modeler about the **style of modeling**:

- **Paradigm** Different paradigms can be used to represent concepts such as time, action, plans, causality, propositional attitudes, etc. For example, one model might use temporal representations based on interval logic while another might use a point-based representation (Chalupsky, 2000). The use of a different "top-level" ontology is also an example of this kind of mismatch.
- **Concept description** These types of differences are called *modeling conventions* by Chalupsky (2000). Several choices can be made for the modeling of concepts in the ontology. For example, a distinctions between two classes can be modeled using a qualifying attribute or by introducing a separate class. These choices are sometimes influenced by the intended inference system. Another choice in concept descriptions is the way in which the is-a hierarchy is build: distinctions between features can be specified higher or lower in the hierarchy. For example, consider the place where the distinction between scientific and non-scientific publications is made: a dissertation can be modeled as `dissertation < book < scientific`

`publication < publication`, or as `dissertation < scientific book < book < publication`, or even as subclass of both `book` and `scientific publication`.

Further, the next two types of differences can be classified as **terminological mismatches**.

- **Synonym terms** Concepts can be represented by different names. A trivial example is the use of the term "car" in one ontology and the term "automobile" in another ontology. This type of problem is called *term mismatch (T or TD)* by Visser et al. (1997). A special case of this problem is the situation in which the natural language that is used to describe the ontologies differ.
  Problems caused by synonyms or different languages are typically approached with thesaurus-based solutions. Usually these problems coincide with semantic problems and require a lot of human effort. Especially, one must be careful not to overlook a concept scope difference (see above).
- **Homonym terms** The meaning of a term is different in another context. For example, the term "conductor" has a different meaning in a music domain than in an electric engineering domain. Visser et al. (1997) calls this a *concept mismatch (C or CD)*. This inconsistency is much harder to handle; (human) knowledge is required to solve this ambiguity.

Finally, there is a one trivial type of difference left.

- **Encoding** Values in the ontologies may be encoded in different formats. For example, a date may be represented as "dd/mm/yyyy" or as "mm-dd-yy", distance may be described in miles or kilometers, etc. There are many mismatches of this type, but these are all very easy to solve. In most cases, a transformation step or wrapper is sufficient to eliminate all those differences.

3.1.3 Discussion

The overview above illustrates that there are many aspects in which ontologies can differ. In principle, all these difference can occur between different ontology versions, although some mismatches are more likely to happen than others. For example, evolving ontologies are usually expressed in one language, so that language-level mismatches will probably occur less frequently than ontology level mismatches. Similarly, we can expect that changes in the concept scope happen more often than paradigm changes, as the latter usually involves a complete re-engineering of the ontology.

We used a number of the categories above for our analysis of current change management strategies (Section 3.3), where we asked for the reasons for changes. In the next chapter, where we describe a framework for coping with ontology change, we will explain how different types of mismatches have different consequences on the interpretation of change.

3.2 Comparison with Database Schema Versioning

Change management is a well known topic from research on database systems. Roddick (1995) gives an overview of the issues that are involved when multiple, heterogeneous

schemas are used for various database related tasks. Some of these issues are also relevant for ontology change management. In this section, we first describe the main issues in database schema versioning. Then, we compare ontology versioning with database schema versioning and explain what the main differences are. Based on this comparison, we list a number of implications for the design of a change management methodology for ontologies.

3.2.1 Database Schema Versioning

In database literature, it is common to distinguish between two variants of schema change, viz. schema evolution and schema versioning. The first is the ability to change a database schema without losing data. The second is a stronger variant: it allows the access of the data in the database through different versions of the schema. Some people from the area of object oriented databases have problems with the use of the term "versioning" for schema changes, as they preserve this term for different instances of an class, and they prefer the term schema evolution. Nevertheless, we will use the terms as explained above.

There are quite a number of *architectural* issues in this schema evolution area, e.g. about the strategy for schema conversion (should the physical schema change) and data conversion (when should the data be converted: just in time or in advance) and about access rights. This type of problems does not seem very relevant for ontology versioning.

Data model issues are more relevant for in our context. Ventrone and Heiler (1991) discuss the effects of changes in the real world on data models. They list a number of examples of domain evolution, ranging from time and unit differences to evolution caused by a "lazy" database administrator that reused a database column for a different purpose. Most solutions that are suggested to cope with this semantic heterogeneity have to do with explicit meta-data with rich semantics. The description of the meaning of the data can then be used to detect and express differences, and to determine the effects of the differences. It can also help to perform translations and conversions. An important observation is that ontologies provide this semantic information *by themselves*.

Another important aspect of database schema versioning is the effect of schema changes on existing data. This is sometimes called "change propagation", and is explored in (Banerjee et al., 1987). This paper describes rules and semantics for schema evolution in object-oriented systems. It does this by specifying a number of invariants, which should hold for every schema. It then specifies rules for handling changes that maintain the invariants. The rules specify whether changes are allowed and whether instance data should be converted. However, this approach is not very well applicable to evolution of ontologies on the Semantic Web, as there is no possibility to reject changes or to force conversions.

The research area of federated databases also shares some issues with ontology versioning on the web. They also discuss development tasks like schema translation and schema integration.

3.2.2 Differences with Ontology Versioning

In this section, we will list the most prominent differences between database schemas and ontologies in general. We then discuss different usage paradigms for database schemas and ontologies. The last group of differences addresses knowledge-representation issues. We only discuss the differences that have direct implications for developing a framework for ontology evolution and versioning.

Ontologies are also used as data

The main goal for schema-evolution support in databases is to preserve the integrity of the *data* itself: how does the new schema affect the view of the old data? Will queries based on the old schema work with the new data? Can old data be viewed using the new schema? The same issues are certainly valid for instance data in ontologies. We can view ontologies as "schemas for knowledge bases." Having defined classes and slots in the ontology, we populate the knowledge base with instance data. However, there is a major second thrust in ontology evolution: ontologies themselves are data to an extent to which database schemas have never been. Ontologies (and not the instance data) are used as controlled vocabularies, to drive search, to provide navigation through large collections of documents, to provide organization and configuration structure of Web sites. And in many cases, an ontology will not have any instance data at all. A result of a database query is usually a collection of instance data or references to text documents, whereas a result of an ontology query can include elements of the ontology itself (e.g., all subclasses of a particular class). Therefore, when considering ontology evolution, we must consider not only the effect of ontology changes on the way applications access instance data, but also the effect of these changes on queries for the ontology contents itself. There is an extra layer of abstraction where database schemas themselves do act as data—metadata repositories (Marco, 2000). Metadata repositories provide the information about various databases and applications in an organization. Ontologies are different from metadata repositories: metadata repositories are designed to store schema and application data, whereas ontologies describe a domain of discourse for any domain. Concepts and relations in an ontology usually have formally-defined semantics that machines can interpret. In addition, metadata repositories are different from schemas themselves, providing an extra layer of description, whereas with ontologies no such extra layer exists. Therefore, while we can learn from the research in the schema-evolution issues for metadata repositories, they will not be directly applicable to ontology evolution.

Ontology data models are often richer

The number of representation primitives in many ontologies is much larger than in a typical database schema. For example, many ontology languages and systems allow the specification of cardinality constraints, inverse properties, transitive properties, disjoint classes, and so on. Some languages (e.g., DAML+OIL) add primitives to define new classes as unions or intersections of other classes, as an enumeration of its members, as a set of objects satisfying a particular restriction. Therefore, any detailed treatment of ontology changes must include a much more extensive set of possible operations.

Ontologies themselves incorporate semantics

Partly because of their richer data model, ontologies usually incorporate more semantics than database schemas. In contrast, database schemas and catalogs often provide very little explicit semantics for their data. Either the detailed semantics has never been specified, or the semantics were specified explicitly at database-design time in the conceptual schema, but this specification was lost in the translation to a physical database schema and is not available anymore. Therefore, with databases, we need specific protocols for resolving conflicting restrictions when the schema changes. These protocols are usually part of a schema-evolution framework (Banerjee et al., 1987). Ontologies, however, are logical systems that themselves incorporate semantics. Formal semantics of knowledge-representation systems allow us to interpret ontology definitions as a set of logical axioms. We can often leave it to the ontology itself to resolve inconsistencies and do not need to do anything about them in the evolution framework. For example, if a change in an ontology results in incompatible restrictions on a slot, it simply means that we have a class that will not have any instances (is "unsatisfiable"). If an ontology language based on Description Logics (DL) is used to represent the ontology (e.g., OIL (Fensel et al., 2000) and DAML+OIL (Hendler and McGuinness, 2000)), then we can use description-logics reasoners to re-classify changed concepts based on their new definitions.

Ontologies are intended for reuse

A database schema defines the structure of a specific database; other databases and schemas do not usually directly reuse or extend existing schemas. The schema is part of an integrated system and is rarely used apart from it. There are exceptions to this rule, which include schemas that support packaged commercial products for applications such as accounting and personnel records. The situation with ontologies is exactly the opposite: ontologies are often intended for reuse and they are not bound to a specific system. Therefore, a change in one ontology affects all the other ontologies that reuse it, and, consequently, the data and applications that are based on these ontologies. Even seemingly monotonic changes, such as additions of new concepts to an ontology, can have adverse effects on the other ontologies that reuse it. If we add a concept that already exists in the reusing ontology, no logical conflicts arise, but the reusing ontology contains two representations of the same concept. We will need to specify an equivalence statement to reflect this fact.

Ontologies are collaboratively developed

Traditionally, database schema development and update is a centralized process: developers of the original schema (or employees of the same organization) usually make the changes and maintain the schema. The development and maintenance of integrated databases (Batini et al., 1986) and federated database systems (Sheth and Larson, 1990) is already much more de-centralized, but at the very least, database-schema developers usually know which databases use their schema. By nature, ontology development (and,

therefore, evolution) is an even more de-centralized and collaborative process. As a result, there is no centralized control over who uses a particular ontology. It is much more difficult (if not impossible) to enforce or synchronize updates: if we do not know who the users of our ontology are, we cannot inform them about the updates and cannot assume that they will find out themselves. Lack of centralized and synchronized control also makes it difficult (and often impossible) to trace the sequence of operations that transformed one version of an ontology into another. Recently, ontologies have become a cornerstone of the Semantic Web (Berners-Lee et al., 2001), which has the model of distributed, reusable, and extendable ontologies at its core. The envisioned huge scale of the Semantic Web and even more de-centralization in ontology development and maintenance greatly exacerbate the problem: in today's Web, we can neither know who uses an ontology that we maintain or how many users there are, nor prevent or require others to use a particular ontology. It is interesting to note that in recent years, the database field is moving in the direction of de-centralization as well: there are standard XML schemas that are reused through different applications, particularly in e-commerce.

Classes can be used as instances

Databases make a clear distinction between the schema and the instance data. In many rich knowledge-representation systems it is hard to distinguish where an ontology ends and instances begin. The use of metaclasses—classes which have other classes as their instances (Chaudhri et al., 1998a)—in many systems (e.g., Protégé (Fergerson et al., 2000), Ontolingua, RDFS (Brickley et al., 1999)) blurs or erases completely the distinction between classes and instances. In set-theoretic terms, metaclasses are sets whose elements are themselves sets. This means that "being an instance" and "being a class" is actually just a *role* for a concept. For example, the "Lonely Planet for Amsterdam" is a specific instance of the class "Travel guides" in a bookstore; at the same time, however, it is a class of which the individual copies of the book are instances. Therefore, analysis of schema-change operations, which considers only effects on instance data, is not directly applicable to ontologies.

3.2.3 Implications for Evolution and Versioning of Ontologies

The differences between ontologies and database schemas that we have outlined above, have direct practical implications for any methodology for ontology evolution and versioning. We discuss the following implications in the rest of this section:

1. The traditional distinction between versioning and evolution is not applicable to ontologies.

2. Defining *what* constitutes compatibility between different versions becomes a more salient issue since there are several dimensions to compatibility (e.g., preservation of instance data, consequence preservation, etc.).

3. The set of change operations that we must consider in classifying effects of ontology changes is much wider. In addition, we must consider the effects of these operations along different dimensions of compatibility

4. We need techniques for determining compatibility between different versions even if we do not have a trace of the changes that led from one version to another.

1 Ontology versioning and evolution is change management

Database researchers distinguish between *schema evolution* and *schema versioning* (Roddick, 1995). Schema evolution is the ability to change a schema of a populated database without loss of data (i.e., providing access to both old and new data through the new schema). Schema versioning is the ability to access all the data (both old and new) through different version interfaces. For ontologies, however, we cannot distinguish between evolution, which allows access to all data only through the newest schema, and versioning, which allows access to data through different versions of the schema. Multiple versions of the same ontology are bound to exist and must be supported. Not knowing how an ontology is being reused means not being able to "force" the reusing ontologies and applications to switch to a new version. Ideally, developers should maintain not only the different versions of an ontology, but also some information on how the versions differ and whether or not they are compatible with one another. For example, the ontology-versioning mechanism in SHOE (Heflin and Hendler, 2000) enables developers to declare whether or not the new version is backward-compatible with an old version (that is, applications and agents can use the new ontology in place of the old one). However, some applications may continue to use the old versions and upgrade at their own pace (or not at all). The management of changes is therefore the key issue in the support for evolving ontologies. Hence, we will combine ontology evolution and versioning into a single concept defined as *the ability to manage ontology changes and their effects by creating and maintaining different variants of the ontology*. This ability consists of methods to distinguish and recognize versions, specifications of relationships between versions, update and change procedures for ontologies, and access mechanisms that combine different versions of an ontology and the corresponding data.

2 Compatibility of ontologies has several dimensions

In order to determine which changes to an ontology are backward-compatible, we need to determine what compatibility means. In databases, backwards-compatibility usually means the ability to access all of the old data through the new schema. In other words, no instance data is lost as a result of the change. For ontologies, query results can include not only instance data but also elements of the ontology itself. Therefore, we cannot express compatibility only in terms of preservation of instance data. Consider a situation in which a new class is added to an ontology as a subclass of an existing class. This change has no effect on instance data and will not change or invalidate answers to existing queries that return only instance data. However, if queries are about the *ontology itself* (e.g., a list of subclasses of a specific class), the answers to existing queries change. This issue becomes even more complicated with ontology languages that support automatic classification (e.g., DAML+OIL): when a class is added to an ontology, a reasoner can re-classify existing concepts and instances, possibly invalidating existing data or applications. When we characterize the effects of change operations we need to take these dimensions into

account.

3 Ontology-change operations and effects

The set of possible change operations for ontologies is larger than the traditional sets of database schema-change operations (Banerjee et al., 1987). There are two causes of the differences between these two sets. The first cause is the richer knowledge model for ontologies: we must add operations that deal with changes in slot restrictions, with slot attachment, and so on. The second cause is the use of *composite operations* which few researchers in the schema-evolution community have addressed (with the notable exception of Lerner (2000)). Consider for example a change in the domain of a slot from a class to its superclass. A model of traffic connections in Amsterdam may have a slot `speed-limit` only for roads. To change the domain of the `speed-limit` slot to include thoroughfares (both roads and canals), we need to "move" the slot up the class hierarchy (imposing a speed limit for boats as well). If we treat this operation as a sequence of two operations, removing the slot from the `Road` class and then adding it to the `Thoroughfares` class, we would have to delete all the values of the `speed-limit` slot for all instances of `Road` after the first operation. However, after the second operation, all instances of Road can have the `speed-limit` slot again. The composite effect of the two operations does not violate the integrity of the instance data, whereas one of the operations does. Therefore, the algebra of ontology-change operations must include these composite operations since their compound effect on schema evolution (1) is predictable and (2) can belong to a completely different class of operations than each of the simple operations that constitute it.

4 There are two modes of evolution

Characterizing effects of specific changes on compatibility between versions of an ontology is important. However, because of the extremely distributed nature of ontologies, we must also account for the fact that we will not always have the trace of changes that led from one version to another. Therefore, we distinguish two modes of ontology evolution: *traced* and *untraced* evolution. Traced evolution largely parallels schema-evolution where we treat the evolution as a series of changes in the ontology. After each operation that changes the ontology (e.g., add or delete a class, attach a slot to a class, change restrictions on slots, etc.), we consider the effects on the instance data and related ontologies, depending on the dimension of compatibility we use. The resulting effect is determined by the combination of change operations.

With untraced evolution, all we have are two versions of an ontology and no knowledge of the steps that led from one version to another. We will need to find the differences between the two versions in an automated or semi-automated way. Rahm and Bernstein (2001) survey the approaches that use linguistic techniques to look for synonyms, machine-learning techniques to propose matches based on instance data, information-retrieval techniques to compare information about attributes, and so on. In the database-schema research, Bernstein et al. (2000) also argue that we can view tasks such as schema mapping and untraced evolution in a similar way. They suggest a formal model for ex-

pressing correspondences between any database schemas, XML DTDs, UML models, and so on. Then any of the tasks for managing correspondences between different sources becomes the task of instantiating such a model.

3.3 Study of Existing Ontology Management Strategies

In this section, we describe how change is managed in existing projects in which large ontologies are maintained. Our research aim is to investigate change management for distributed ontologies (see Chapter 1), however, the projects described in this section involve centrally maintained ontologies. With the analysis of change management strategies for centralized ontologies, we achieve two goals. First, it provides us with an overview of current techniques, which might be also applicable for distributed change management—possibly in an adapted form. Second, if compared to the goals of ontology change management, we get an understanding of possible shortcomings of the current strategies.

We start with a description of the design of the study in the next section. The subsequent sections consists of reports of the interviews held. The results of the interviews are represented via narrative description of the issues that were discussed. In Section 3.3.6, we summarize our observations and look forward to the requirements for distributed ontology versioning.

3.3.1 Research Design

As research method, we used the "semi-structured interview" (Robson, 2001). Before we started the research, we have created a guideline for the interviews. In this guideline, we go through all aspects that are mentioned in the section about the aims above. During the interviews, we started from the questions in the guideline, but we did not stick to it exactly. If necessary, we asked additional questions when seemingly interesting issues popped up that were not covered in the interview, we sometimes changed the order of the questions when the course of the conversation asked for it, and occasionally we left out questions that were clearly not relevant. The guideline as we used it is printed as Appendix A. The questions in the interview are based on our experience with change management and on an examination of several versions of some of the selected projects beforehand.

Aims

The overall goal of the interviews is to understand how and why changes in ontologies occur and what processes exist to manage the changes. In the interviews with the maintainers of the ontologies, we are looking for facts, behavior and beliefs. Examples of the facts we are looking for are: what is the ontology about, how is it structured and developed, how is it used, what kind of changes occur, do these changes hamper the usage of the ontology, and—if yes—in what sense? Examples of behavioral aspects that we want to discover are: how are changes handled, what is the decision process with respect to the changes? The major beliefs that we wanted to reveal are about the main problems caused

by ontology change and about the kind of support from a versioning methodology that would be most useful.

Selection of Projects

We have selected projects in which relatively large ontologies are maintained, that have multiple users and that already have some history. A secondary criterium was the accessibility of the persons that maintained the ontologies, as we preferred to have face-to-face interviews with the maintainers. All together, we have talked with the maintainers of four different ontology projects, two in Stanford CA, USA, one in Berkeley CA, USA, and one in Amsterdam, the Netherlands.

3.3.2 PharmGKB

The PharmGKB ontology[1] is part of the PharmGKB project (Klein et al., 2001b), a knowledge base for pharmacogenetics and pharmacogenomics. The knowledge base is a central repository for genetic and clinical information about people who have participated in research studies at various medical centers in a collaborative research consortium. In addition, genomic data, molecular and cellular phenotype data, and clinical phenotype data are registered from the scientific community at large. Its aim is to aid researchers in understanding how genetic variation among individuals contributes to differences in reactions to drugs.

At the time of the interview[2], the number of users is estimated at 100. Because there are no registered users, the estimation is based on the log of the web server. There is a lot of instance data in the knowledge base, however, technically speaking the instance data is not connected to the ontology anymore. For performance and reliability reasons the data is stored in a database. The schema for this database is not derived from the ontology; instead, it is based on an analysis of the instance data that had to be stored. In the future, the ontology might be used again for inference tasks, for example, data validation, finding erroneous data and inferring new drug–gene relations. This will possibly be implemented via a “database to knowledge base push”.

The ontology consists of different parts. The first part is taken from MeSH[3], a second part is the drug ontology from Apelon[4], the third segment is the PharmGKB specific part. The first two components are imported, the third is created in an iterative process with domain experts. The different parts are virtually integrated, i.e. in the user interface they seems to be seamlessly integrated, but in the database, the parts are stored in separate tables. All data resides in the database and the maintainers are not aware of any other usage of a (part of) the ontology. They do not expect that reuse is easy, as ontologies are often very closely matched to the purpose of a specific project, and the granularity might

[1] See `http://www.pharmgkb.org/`

[2] The interview was held in November 2002. In the meanwhile, the project has evolved further. The description in this section is not an adequate representation of the current situation (especially the statistics are outdated), but it can still serve as a source for change management techniques.

[3] Medical Subject Headings, a hierarchical vocabulary thesaurus, see `http://www.nlm.nih.gov/mesh/`.

[4] `http://www.apelon.com/`

not be correct. For example, the Apelon ontology goes down to doses of drugs, whereas the PharmGKB just needs the drugs. For vocabularies, such as in GO and MeSH, reuse might be easier.

The development of the ontology started three years before the interview and has been a four-person enterprise. For additional domain knowledge, 3 to 5 domain experts from the medical centers were consulted during the development phase. This phase lasted about six months. In this phase changes to the ontology occurred every month or every other month. There were incremental changes to the data model and refinements. After this phase, the model settled down and since it has been translated into a database, it has been more or less stable. Occasional changes are first performed in a beta database and later on rolled over to the production database. Data is continuously added to the database. There is no formal validation process for data additions, but people who submit data can go to the web site to see whether their data are correctly stored. There are several verification methods performed when data is added to the system. For example, the data has to be correct XML, there are Schematron rules[5] that enforce the structure, and the database does some verification. Higher-level algorithms to validate the data are being developed. The way the ontology was divided into different parts has never changed.

Different releases of the ontology are identified by version numbers, which are just incremental numbers. This was the case for releases of the knowledge base and is now the case for database releases. There is a release about every 6 or 8 months. The releases are more or less dictated by feature requests and additional functionality needs. There is no roll-back facility (i.e. a mechanism to roll-back the whole database to a previous version), but there is a mechanism for rolling over new changes from the beta database to the production database. This procedure works as follows: the changes to the ontology (and database schema) in the beta database are restricted to changes that are "non-destructive for data"; at the same time, the production database is actively used; because all changes to the production database are additions of instance data, a roll-over can be performed by combining the schema from the beta database with the instance data of the production database.

There are three types of releases. The first type are releases caused by changes in the database, as described above. Then there are releases caused by changes in the web functionality, for example, in the format of the XML files. There is a formal process for such changes, with a one month comment period etc. Third, there are also display changes, which requires that other things are adapted as well, for example the queries to get the information that will be displayed. The last two types of releases often coincide, but not always.

Specific motives for changes in the ontology were data model changes, new kinds of data, granularity changes, and changes in display requirements, i.e. changes in the functionality of the graphical user interface for uploading and querying data. In the early phase of the project most changes were restructurings of the domain. After that there was a phase in with many extensions to the domain. In the latest phase, there are mainly corrections. Another type of change that is important in this project are changes with respect to instance data, both modifications of instance data and additions of instances.

[5] `http://www.ascc.net/xml/schematron/`

The study centers not only submit new data, but they also want to update it or retract it.

The change management strategy has been developed during the evolvement of the project. The most specific problem that the evolution caused was that the referential integrity was broken by the deletion of instances. The project recently changed to a process in which nothing is deleted anymore, but only "retired". When the strategy and the software was in place, no problems were encountered anymore.

With respect to the support that is required from versioning systems, the maintainers make a distinction between the development phase and the production phase. During the development phase, things like consistent reasoning, synchronization, data translation, propagation support are important. During production, it is important to have referential integrity and to know the effect on future functionality. For example, is it still possible to ask a specific query or display specific information.

3.3.3 EMTREE thesaurus

EMTREE[6] is a thesaurus of medical terms and drugs developed and maintained by Elsevier (Elsevier/Embase, 2003). Its goal is to relate information to index terms in a consistent way, for example to index abstracts of articles. It is used for subject indexing in EMBASE, a database with over 9 million citations of literature on human medicine and related disciplines. Besides that, it is used by other vendors to index articles; some are behind with updating and thus use an older version of EMTREE, others have included additional information.

EMTREE is a "'poly-hierarchically" structured, which means that there are multiple orthogonal hierarchies that classify the terms. Each term contains a list of synonyms and a number that is based on its place in the hierarchy. It incorporates MeSH terms used by the National Library of Medicine, as well as almost 20,000 CAS registry numbers assigned by the Chemical Abstracts Service. EMTREE includes more than 45,000 drug and medical terms, 10,000 numeric codes and over 190,000 synonyms. The thesaurus is split into 15 parts, called facets. This partition is mirrored from MeSH. The development of the ontology started in 1987, before that it was flat list of synonyms. There are three people working on the ontology, plus occasionally a few others. These three people do the modeling and provide the content knowledge. These developers usually work on different facets of the ontology.

There is a production version and a development version of EMTREE. Usually at the beginning of a year, the development version becomes the new production version. There is no versioning procedure in the sense that multiple versions are maintained in parallel, only the latest version is used as active version. The partitioning of EMTREE has never changed. The hierarchy does not change very often, most changes are additions of new terms. The indexers, i.e., the people that use the thesaurus to index articles, suggest these changes via paper forms during the year. If accepted, the changes are performed in the development version and will appear in the next production version. Because of the yearly release schedule, it takes between 3 and 15 months before a suggestion for change is incorporated in the production version. To allow indexers to use the terms

[6] See `http://www.elsevier.nl/homepage/sah/spd/site/`.

earlier, a suggested new term is directly added to the *production* version as a *candidate* term. These terms can be used by the indexers, but there is no guarantee that they will appear in the next production version.

There are in total four different states for terms. Besides *candidate*, terms can be *preferred*, *synonym* or *de-active*. Preferred terms are used to index the articles and abstracts. Synonym terms are synonyms of preferred terms and are only used to interpret queries, not to index articles. De-active terms are deprecated terms that are not used anymore. All de-active terms stem from before 1987, when the thesaurus was still a flat list; all terms that had a frequency of less than 60 were flagged as "de-active". Sometimes synonyms become preferred term and vice-versa. Consequently, some articles may be indexed with terms that are not preferred anymore. Also, some articles are indexed with candidate terms that never made it to the thesaurus, and a small amount of older articles are indexed with de-active terms. In 1998, the complete MBASE article database was updated by replacing all synonyms that were used as index terms by their preferred term.

Change suggestions are judged by a special department. The two main reasons for not including new terms are that the terms are not correct, or that they occur too infrequently. Synonyms of existing terms are often accepted, e.g. spelling variants or new names. The main reasons for changes in the terms are the introduction of new names for existing chemicals, the invention of new chemicals, and the correction of errors. Occasionally the hierarchy is restructured because of new insights in the field. Another class of changes is introduced by changes in MeSH, because of the mapping that exists between MeSH and EMTREE. Occasionally terms are deleted from the index, if they are absolutely incorrect. More often, such terms become synonyms of other terms.

Besides the concepts mentioned above, each term in the EMTREE thesaurus also contains a type attribute and an extensive history of changes, in the form of former relations and time-stamps of additions.

3.3.4 EON ontology

The EON project[7] seeks to create an architecture that developers can use to build robust decision-support systems that reason about guideline-directed care. Within this project, an ontology of the guideline domain is being developed (Tu and Musen, 1999). The goal of this ontology is to come up with a structure in which guideline knowledge can be encoded. This knowledge is used for different tasks: to interpret patient data in the concepts of the guideline, to encode decisions, i.e. to prescribe the different steps that have to be taken, and to specify how tasks can be refined.

There are two physicians in a hospital who use the ontology to encode medical knowledge. They are creating a knowledge base about hypertension as instance data of the ontology. The created knowledge base is part of a clinical decision-support system that was installed at 8 geographical locations and used by about 100 physicians (Goldstein et al., 2004).

The development version of the ontology is split up into several modules, but in the production version everything is aggregated into one large project. The partitioning

[7] `http://smi-web.stanford.edu/projects/eon/`

in different modules is partly based on domain aspects and partly on implementation issues. For example, one module is called “BaseQueryKB”, others are “medical domain”, “time”, “guideline” and “EPR” (a simple ontology of an electronic patient record). The development of the ontology started around six years ago. There is one person who does the modeling of structural concepts in the ontology, but sometimes new ideas come in via collaboration with other people. The part of the ontology about the medical domain is developed by the two physicians.

The project has been in development phase for a long time, in which there were major changes. Since it went into production phase, there have only been incremental changes. There has been a very long test period, about 1 year, and the developers were quite confident about the ontology. The structural part of the ontology did not change much in the last half year before the interview. The two physicians ask the developer of the structural part to do specific changes if necessary. The changes in the structural parts are indirectly validated because the knowledge base is extensively tested. Changes are made in a non-production version first, and later on rolled over to the production version. Version management is done at file level, by checking in the changed projects into SourceSafe,[8] a file versioning system. Each change is checked in into SourceSafe and is given a label, usually the date. There is no pre-defined release schedule. According to the developer, change management was a huge problem. Because the two physicians did not have access to the SourceSafe database, files had to be exchanged back and forth, with explicit understanding of who was making changes at each moment in time.

The developer of the EON ontology distinguishes several reasons for changes. One of these is the addition of medical knowledge, e.g. a specific laboratory result. Also, sometimes abstractions area created, e.g. instead of the general term “drugs” a specific set of drugs. Sometimes the change is caused by the introduction of a new type of knowledge. An example is the categorization of drug doses into ”low,” ”medium,” and ”high” doses. New structures had to be built for this type of categorization. This happened at several occasions. A completely different reason for changes is the introduction of additional functionality / modeling capabilities of the tool, i.e. the possibility to specify constraints. This resulted in several additions to the ontology. Yet another reason for change is extending the *competency* of a ontology, i.e. the extent of the task for which an ontology is used. For example, the concept of “user” was introduced to assign tasks to different classes of people. Therefore, the organizational aspect had to be modeled. Finally, another cause of changes in EON was the introduction of new inference mechanisms, for example the ability to reason with constraints. This required things to be modeled in a different way.

The major problem that the developers encountered during the development of the ontology was finding the right balance between making the ontology suitable for a specific task, and keeping it generally applicable. Each change request had to be judged on these aspects. Keeping the knowledge base consistent with the ontology during the evolution was another problem. With respect to desired support for versioning tasks, the developer mentions several things. One thing is help in finding out which subsequent changes are necessary in both instance data and in other projects, as result of a change,

[8] See `http://msdn.microsoft.com/ssafe/`

i.e. the repercussion of changes. Also, the developers say that a high level log of changes would be helpful for the purpose of understanding changes. Other issues are: being able to work with multiple users on one ontology, and having technical support for working with ontology modules, e.g. moving concepts between modules.

3.3.5 Gene Ontology

The goal of the Gene Ontology (Ashburner et al., 2000) is to provide a controlled vocabulary that can be applied to all organisms, even as knowledge of gene and protein roles in cells is accumulating and changing.[9] The initial goal in 1998 was to allow researchers to query related databases and get related proteins and gene products. At that time the project was a collaboration between three model organism databases: FlyBase (Drosophila), the Saccharomyces Genome Database (SGD) and the Mouse Genome Database (MGD). Since then, many other databases have been included in the project. The ontology consists of three structured, controlled vocabularies that describe gene products in terms of their associated *biological processes*, *cellular components* and *molecular functions* in a species-independent manner. The data in the different collaborating databases can be seen as instance data of the ontology. Besides developing the ontology, the Gene Ontology (GO) consortium also makes cross-links between the ontologies and the genes and gene products in the collaborating databases, and develops tools that facilitate the creation, maintenance and use of ontologies.

There is one central place were the ontology resides, but the control is distributed. There are a few full time "curators" that work on the vocabulary and the relations. GO users can make suggestions for additional terms or for other improvements via change requests. This is implemented in a system that allows the submitter to track the status of their suggestion, both online and by email, and allows other users to see what changes are currently under consideration. Eventually, the curators perform the changes. Changes occur on a daily basis. Over time, GO has grown larger, has become more complex, and has grown in breadth. A record is kept of every change, and there are procedures that check for local conflicts in the ontology. Initially, the changes were managed via CVS, later on a database was used. The database allows full roll-backs of changes. Every month a new version of GO is published. This release is a snapshot of the current status of the database.

The coordinator of the project mentions several reasons for changes. One reason is the strive for completeness: incomplete parts of the ontology have to be completed. Also, errors have to be repaired. A special reason is the splitting of compound terms. This normally involves creating a term, obsoleting the old term, and creating equivalence relations. However, most of the changes are additions. There were two major problems caused by the evolution of the ontology. The first problem is that it was not possible to track a term when it disappeared from the ontology. This is now solved by obsoleting terms instead of deleting them. A second problem is mis-annotation. It happens that people have labeled data with a wrong assumption, probably because of an ambiguous definition. When the definition of the term is clarified there is no mechanism that triggers

[9] `http://www.geneontology.org/`

people to verify the way in which they used the term. This problem is solved by a policy that prescribed that definitions may not be changed; if there is a different meaning, a new term has to be introduced with a different ID.

3.3.6 Discussion

When we look at the ontologies and the change management strategies that are described in the previous sections, we can make a number of observations.

First, we see that none of the described projects have a real distributed development model. To control the changes, the projects either have a central authority that makes decisions about the changes, or they have divided the ontology in different parts and distributed the control accordingly. This works when the different developers work closely together, or when the parts are only weakly connected.

Many projects try to minimize the number of changes that have to be made. In order to achieve this, most projects started with a long development phase. During this phase it was less problematic to make changes, either because there was not much instance data, or the development was very centralized. In the production phase, some projects even restricted the possible types of changes.

One of the projects, i.e. the EMTREE thesaurus, had a change management procedure that deliberately sacrificed a bit of the completeness in favor of usability, with the inclusion of candidate terms that may disappear later on. The completeness and correctness of the procedures seemed to be a basic requirement in the other projects.

For storing and retrieving the different versions, it turns out that there is nothing specific used for ontologies, but that all projects fall back to well known techniques. Most of the described projects used a database, one used a file based system.

The reasons for change that are mentioned most often, are extensions of the coverage of an ontology, and changes in the way a domain is interpreted (conceptualization). These two reasons are mentioned in three of the four projects. Corrections of errors and changes in use of the ontology were also mentioned more than once. Surprisingly, “changes in the domain” was only mentioned once. A possible explanation for this is that two of the four projects are in a relative early state, in which more attention is paid to the growth of the ontology than to the validation of the ontology.

In most of the interviews, people seemed to find it difficult to talk about the kind of change support that they would like to have. It attracts attention that the people that represented the “more developed” projects, i.e. EMTREE and GO, were not able to mention any desired support at all. A hypothesis for this is that these projects already have developed working strategies for the problems that they encountered.

Table 3.1 gives an overview of the main issues in the interviews and whether or not they were mentioned for a specific project. If the table contains an empty cell, it does not necessarily mean that the specific issue is not the case for the project, but just that it was not mentioned during the interview.

Concluding, we can say that there are several techniques and strategies that can be applied to prevent that the evolution of ontologies will cause problems. However, many of these techniques are easier to apply in a (relative) centralized setting, than in a distributed setting. For example, restricting the possible changes, and partitioning the development

	PharmGKB	EON	EMTREE	GO
Change management strategies				
development / production phase	x	x	x	
restricting changes	x			x
working on different parts			x	x
central authority that validates		x	x	
provisional changes			x	
Means for version management				
file-based version management		x		
database version management	x		x	x
Reasons for changes				
change in conceptualization	x	x	x	
world / domain changes			x	
correction of errors			x	x
extension of the domain	x	x		x
granularity changes	x			x
change in usage	x	x		
Desired change management support				
referential integrity	x			
prediction of effect on tasks	x			
propagation of changes		x		
multi-user editing capabilities		x		
high-level change representation		x		

Table 3.1: Overview of the main issues in interviews.

in different parts is difficult to achieve when there is no control of the development. The study did not give a clear answer to the question what a change management method for ontologies should achieve. In the next section, we will list a number of goals that people might want to achieve with respect to evolving ontologies.

3.4 Ontology Evolution Tasks

In this section, we describe a number of different tasks for which users that work with evolving ontologies might need help. This can be seen as a list of aims for a versioning methodology for ontologies. This list is partly based on the study described in the previous chapter, and partly on our general experience with the use and development of ontologies.

The general aim of ontology evolution methodology would be to achieve interoperability between versions. In this case, interoperability can be defined as the ability to use different versions of ontologies and possible data sets together in a meaningful way. Meaningful means that the way in which ontologies or data are used is not conflicting with the intended meaning of the ontology.

However, this general goal can be achieved in several ways and has several specific interpretations, which may differ for the different tasks for which ontologies are used. The list is not meant as a definition of components of a versioning support system, but it sketches the wide area of tasks where some support for change management might be

useful.

Data accessibility

The most often mentioned objective of versioning is the ability to access instance data via a different version of the ontology than the one that was used to describe it. In this case the interpretation of interoperability is data accessibility. Data accessibility can be achieved in two ways:

1. Restricting the modification to changes that do not affect the interpretation of the data, i.e. *backward compatible* changes. In this case the change in the ontology does not change the perspective on the data (Heflin and Hendler, 2000).

2. Translating data structures from one to another version. For example, if two classes are merged into one, this usually has as a result that instances of the original classes cannot be accessed anymore. However, if the querying agent knows about the merge, it can translate the instances of the original classes to the new class.

Full data accessibility is not always achievable, of course. Some changes might make a part of the data inaccessible, for example, a deletion of a class.

Data translation

A related goal is the translation of data sources that conform to one version of an ontology in such a way that they conform to another version. This requires that consecutive changes between the two versions of the ontology are executed in the data sets. This is useful in situations where there is control over the data sources and it is important to keep data sets up to date with the ontology, e.g. in a company where the information on the intra-net has to be annotated with a specific ontology.

Consistent reasoning

Another interpretation of interoperability is that the reasoning over the ontology is not affected. This forms a second objective of versioning. In general, this would mean that answers to logical queries are not changed by modifications in the ontology. Of course, this is almost never true in the general case: only non-logical changes will have this effect. For each logical change a query can be invented that will have a different answer.

However, it might be very useful to know that a change in an ontology does not affect the answers to a *specific set* of queries. In many cases, applications will not ask arbitrary queries to an ontology, but only a number of predefined ones. For example, an application might be interested in the instances of a class, or in the properties that can be applied to a class.

If it can be predicted whether or not the answers to such a set of queries will change, it is possible to decide if the versions of ontologies can be exchanged for a specific task.

A set of queries which should give an consistent answer can be defined explicitly, by listing those queries, or implicitly, by specifying the type of reasoning that is assumed.

For example, for ontologies that have an underlying description logic, consistent reasoning means that the derived subsumption hierarchy is not affected by the changes in the ontology.

Synchronization

Something else that is often considered to be part of versioning support is the ability to make copied (and possibly changed) versions of ontologies up to date with a remotely changed ontology. This might be desirable when an ontology is copied and the original is consecutively changed but it is still necessary to work with the original one, too. This requires that all consecutive changes in the original ontology are carried out in the local copy. This process is called *synchronization* in (Oliver, 2000). The term used in software development is *patching*.

This goal is often part of another objective: collaborative development of ontologies. Besides synchronization, collaborative working requires mechanisms for access control (e.g. locking) and conflict resolution.

Management of development

Something else that a versioning framework could support is a step by step verification and authorization of changes to an ontology. This task is also useful in collaborative working scenario. It is related to *synchronization*, but there are some differences. First, it allows a step-by-step acceptation or rejection of the changes, so the user can control the process. A practical difference is that the changes are often performed to the unchanged ontology, instead of to a locally changed ontology. As a consequence, less conflicts will occur in this scenario.

This task requires that the changes in an ontology are presented to the user in a such way that he or she can understand what the change is. This means that it is often not enough to show a list of additions and deletions of terms, but as “aggregations” of a number of atomic changes that are performed for the same reason. For example, a change that occurs quite often is the introduction of an additional distinction: a class gets two new subclasses and its former subclasses become subclasses of the new ones. It makes much more sense if this is presented to the user as one operation instead of 5 or more atomic addition and move operations. This part of the task was mentioned in the interviews as “high-level visualization”.

Editing support

Another type of change management support is support during the editing of changes in ontologies. One aspect is the propagation of changes, i.e. the consequences of effects of changes in other parts of the ontology. This can be as simple as highlighting the consequences, or as advanced as automatically performing necessary additional changes. For example, concepts and properties could be updated or deleted automatically if they refer to other deleted concepts. An implementation of this type of support is described in (Stojanovic et al., 2002). They use different “strategies” that describe the what actions

should be performed if concepts are deleted. Such kind of strategies could also be applied to preserve particular properties of model, for example its logical consistency, or—in case of OWL—the specific sub-language it uses. In the RDF storage and querying engine Sesame (Broekstra et al., 2002a) a similar system is implemented to keep the deductive closure of the RDF Schema model consistent when statements are deleted (Broekstra and Kampman, 2003a).

3.5 Discussion

In this chapter, we have seen that changes in ontologies can make two versions different in various aspects. We then discussed what makes ontology change management different from database schema versioning. It turned out that because of the inherently distributed nature of ontologies and the higher expressivity, other means are necessary. From the study of current ontology development projects, a similar picture arose: the strategies used work best in relatively centralized environments. In the next chapter, we will sketch a framework of ontology change that combines different methods and techniques, considering the additional requirements that a distributed setting brings with it. The framework aims to provide elements that take the sketched ontology evolution tasks into account.

Part II

Framework

Chapter 4

Framework for Ontology Evolution

***Note**: An earlier version of the framework described in this chapter is published in the proceedings of the IJCAI 2003 workshop on Ontologies and Distributed Systems (Klein and Noy, 2003).*

The description of the idea and mechanics of the Semantic Web (Chapter 2) and the analysis of current practices in change management for semantic models (Chapter 3) lead to a number of general guidelines for the design of an architecture that copes with the evolution of ontologies in a distributed setting. In this chapter we outline a framework for ontology evolution. We first introduce our vocabulary and describe a number of basic assumptions behind our approach (Section 4.1), then we explain the basic elements of the framework and the relations between them (Section 4.2), and finally we sketch how these elements can be operationalized. In the next two chapters we discuss the elements in detail.

4.1 Vocabulary and Assumptions

In our framework we assume that there are online, distributed ontologies without a central authority that publishes or authorizes ontology versions. Everybody can publish, reuse, extend ontologies and relate to other ontologies on the Web. The ontologies consist of definitions of classes and relations between them. We do not assume that a specific ontology language is used.

4.1.1 Conceptualization vs. Specification vs. Representation

An assumption behind our ontology evolution framework is the fact that an ontology is not just a digital specification of definitions, but a representation of a *human* construct.

This assumption has an important consequence for the interpretation of the meaning of changes. To make this clear we look at different levels at which an ontology can be interpreted.

Levels of Interpretation of an Ontology

According to the often mentioned definition of Gruber (1993), an ontology is a *specification* of a *conceptualization*. This specification is usually *represented* in some ontology language. Therefore, an electronic file containing an ontology, or even a piece of paper with an ontology, can be interpreted at different levels. Figure 4.1 shows a distinction between different levels of interpretation of an ontology as an UML class diagram.

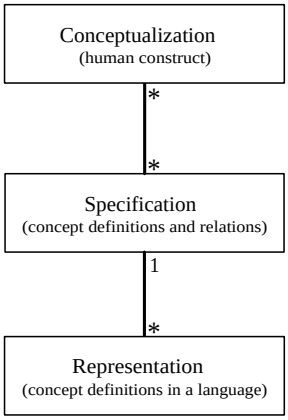


Figure 4.1: Three interpretation levels of an ontology.

At the conceptualization level is the actual interpretation of a specific domain by a (number of) human(s)[1], which basically is an opinion about the important concepts and their relations. For example, a conceptualization about vehicles could contain the different types of vehicles and their characteristics.[2]

The specification level contains precise definitions of the concepts and the relations between them. In the aforementioned example, the specification could distinguish between "cars", "motorbikes" and "bikes", and describe them by their properties, for example the way they are driven (by engine or human power), the number of wheels, the things they can transport etc. There is often more than one way to specify a conceptualization. Therefore, the actual specification of concepts and properties is only one of a number of possible specifications. For example, a "motorbike" could be defined as specific type of motor-vehicle, or as a bike with a motor. Both specifications could refer to

[1] Many people consider it as essential that an ontology is a *shared* view on the world (Borst, 1997), although a more pragmatic opinion, in which an ontology is just a personal view on concepts and their relations, also exists (Yan et al., 2003).

[2] As a conceptualization is just a "interpretation", we cannot give a more concrete example without also providing its specification.

the same concept. Also, a not very detailed specification could possibly refer to different conceptualizations.

The representation level is the actual formalism in which the specification is expressed. Simply said, different representations coincide with different ontology languages. As each specification can be expressed in different languages, there are multiple representations for each specification. However, a specific representation refers to only one specification, because we assume that the ontology languages are capable of providing an unambiguous and complete representation of the concepts and relations in the specification.[3] The textual description of a "motorbike" in the previous paragraph is a example of a representation; another example is the following description in RDF Schema.

```
<rdf:Class rdf:ID="Motorbike">
  <rdf:subClassOf rdf:resource="#Motor-vehicle">
</rdf:Class>
```

Types of Changes

We define an **ontology change** as an action on an ontology that results in an ontology that is different from the original version. However, based on our distinction between different interpretation levels of an ontology, it is useful to distinguish between different types of ontology changes. We will use these terms in the remainder of this book.

- **conceptual change**: a change in the conceptualization;
- **specification change**: a change in the specification of a conceptualization;
- **representation change**: a change in the representation of a specification of a conceptualization.

Because there are no one-to-one relations between the conceptualization, the specification, and the representation of an ontology, changes in one interpretation level are not always changes in the other interpretation level.

A change in the specification does not necessarily coincide with a change in the conceptualization, and changes in the specification of an ontology are not per definition ontological changes. For example, there are changes in the specification of a concept which are not meant to change the concept itself; think of attaching a property "fuel-type" to a class "Car". Both class-definitions still refer to the same ontological concept, but in the second version it is described more extensively. Theoretically, the other way around is also possible: a concept could change without a change in its specification. However, this usually means that the concept is modeled in a very superficial way.

To distinguish between changes in ontologies that affect the conceptualization and changes that do not affect it, Visser et al. (1997) uses the following terms:

[3] Note that, in practice, a specification can not exist without a specific representation. Therefore, the specification is bound by the expressivity of the language, which makes our assumption about the completeness and unambiguity of the languages realistic. Still, it is possible to have multiple representations of a specification. In that case, the specification is bound by the overlap of the expressivity of the languages involved.

- **conceptual change**: a change in the way a domain is interpreted (conceptualized), which results in different ontological concepts or different relations between these concepts;
- **explication change**: a change in the way the conceptualization is defined, without changing the conceptualization itself.

It is not possible to determine automatically whether a change in the specification is a conceptual change or an explication change. This requires insight in the conceptualization, and is basically a decision made by the ontology engineer. However, heuristics can be applied to suggest the effects of changes on the conceptual relations in the ontology. This mechanism will be worked out in Chapter 6.

Second, a change in the representation is not always a change in the specification. If an ontology is translated into a different language with an equivalent or larger expressivity, the specification doesn't change. If the expressivity of the target language is more restricted, then the change in the representation implies a change in the specifications. Because of the cardinality of 1 of the relation between the representation and the specification in Figure 4.1, a change in the specification is assumed to imply a change in the representation as well.

A representational change is typically a change at the syntactic level. For example, in RDFS there are two ways to define classes: via an explicit `rdf:type` statement, or via the shorthand notation `<rdf:Class ...>`. Changing from one to the other definition is a representation change.

Note the difference between Visser et al.'s *explication change* and our *specification change*. Our definitions simply refer to changes in one of the interpretation levels, while the *explication change* says something about the effect of a change in one level on the other level. In our terms, Visser et al.'s *explication change* would be defined as a "specification change without a conceptual change".

4.1.2 Task Dependency

The task dependency of the effects of change is yet another assumption. It is not possible to describe the effects of change operations on the compatibility of two ontologies in general terms (e.g., "compatible" or "not-compatible"), but only by referring to a specific use case of the ontology.

This is notably different from databases, where "backward compatible" usually means that one can access old data correctly via the new version of the schema. This is also the perspective that is described in (Heflin and Hendler, 2000), and is an important perspective in the context of the Semantic Web. However, for ontologies there are also other perspectives, resulting from the different tasks for which ontologies are used.

For example, suppose we have a small ontology with a class "Conference" and a class "City" connected with a property "located_in" and a small data set `IJCAI03 located_in Acapulco` (see Figure 4.2). We can now derive that `Acapulco` is an instance of "City". Now, let us change the ontology in such a way that "City" gets a superclass "Location" and the domain of "located_in" is changed to "Location". This is a backward compatible change from the data accessibility point of view, i.e. we can still access all data via the

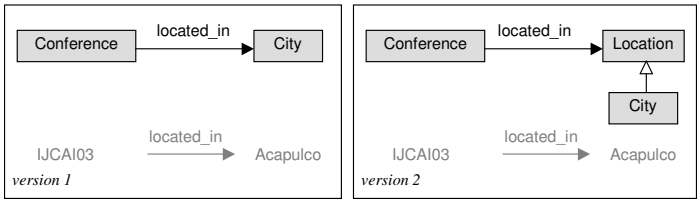


Figure 4.2: A change in a small ontology.

ontology, and the ontology still describes the data correctly. However, if we now query for instances of "City", we do not retrieve `Acapulco` anymore, as it is only known as instance of "Location".

The fact whether a change is compatible or not depends on what is to be preserved. In the example above, this was either the data itself, or the answers queries for instance data. The following perspectives are possible:

- data preservation—no data is lost in transformation from the old version to the new one;
- ontology preservation—a query result obtained using the new version is a superset of the result of the same query obtained using the old version;
- consequence preservation—if an ontology is treated as a set of axioms, all the facts that could be inferred from the old version can still be inferred from the new version;
- consistency preservation—if an ontology is treated as a set of axioms, the new version of the ontology does not introduce logical inconsistencies.

In Chapter 6, we discuss a number of specific variants of the first two perspectives in detail.

Thus, te effect of a change can not be described in general but should always be related to a specific usage scenario. A particular change can then be associated with different consequences for different types of usage. In example above, we could specify for the class "City" that there is no effect on data accessibility, but that the effect on a query for the instances is that some of the previous instances cannot be found anymore.

4.1.3 Different Change Representations

Changes between two ontology versions can be represented in a number of different ways. Each of these ways provides different information at a different level of detail. Among others, we can find the following change representations. For two version V_{old} and V_{new} of an ontology, we can have:

ontologies only: the old version V_{old} and the new version V_{new} of the ontology; this provides no explicit change information, but can be used as a basis to find other change information;

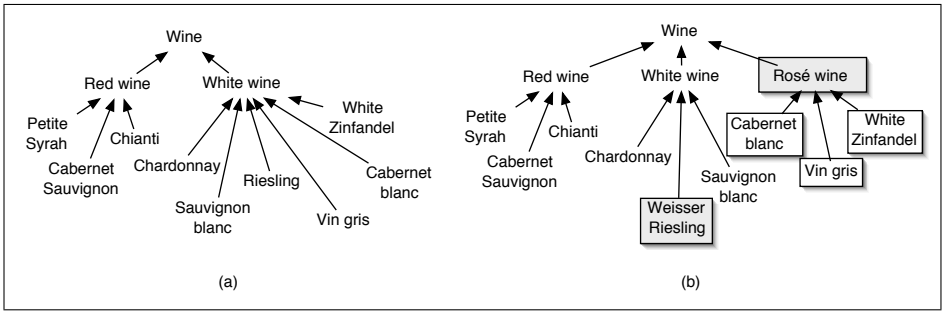


Figure 4.3: Two versions of an ontology (a and b).

log of changes: a record of the changes as they are performed; this is list of changes that applied to V_{old} results in V_{new};

structural diff: a mapping between concepts and properties in one version and their counterparts in the new version, together with a list of added and removed concepts;

conceptual relations: an explicit (but possibly partial) specification of the conceptual relations between concepts in V_{old} and corresponding concepts in V_{new}.

These representations are illustrated in the following example. Suppose that we are developing an ontology of wines. In the first version (Figure 4.3a), there is a class Wine with two subclasses, Red wine and White wine. The hierarchy also includes some specific types of red and white wines. Figure 4.3b shows a later version of the same ontology fragment. Note the changes: there is a new subclass of Wine, Rosé wine; the classes that were previously subclasses of White wine—Cabernet blanc, White Zinfandel, and Vin gris—are now subclasses of the new class Rosé wine; the Riesling class is renamed to Weisser Riesling.

One of the easiest change representations to create, with the appropriate tool support, is a **change log** between versions. A change log records an exact sequence of changes that occurred when an ontology developer updated V_{old} to arrive at V_{new}. Many ontology-editing tools, such as Protégé [4], OntoEdit (Sure et al., 2002) and others, record changes that developers make. There are several detailed proposals for the information that logs should contain (e.g., versioning in KAON (Stojanovic et al., 2002; Maedche et al., 2003), CONCORDIA (Oliver et al., 1999)). For example, the evolution framework of KAON provides a number of "add", "set" and "delete" operations. The log contains a list of specific operations, such as "AddPropertyDomain" or "RemoveSubConcept" with references to the concepts or properties that they operate on.

Most logs of ontology changes are quite similar to the KAON format. They contain simple ontology changes, where the level of granularity at which changes are specified

[4] The Protégé project, see `http://protege.stanford.edu/`

is close to a single user-interface operation. A log is that it provides a complete and unambiguous change specification at a very fine level of detail. Figure 4.4 shows a possible log of changes between versions from Figure 4.3.

```
Feb 25 13:36, user,
    changeName oldName=Riesling, newName=Rheinriesling
Feb 25 13:37, user,
    changeName oldName=Rheinriesling, newName=Weisser Riesling
Feb 25 13:37, user,
    addSuperclass child=Cabernet blanc, parent=Rosé wine
Feb 25 13:37, user,
    removeSuperclass child=Cabernet blanc, parent=White wine
```

Figure 4.4: A fragment of a log of changes for the example in Figure 4.3.

Change logs may not always be available, however. In a dynamic and de-centralized environment such as the Semantic Web, we may have access only to the old and the new version of an ontology, but not to the record of the change. Furthermore, change logs are less useful in an environment where several editors update an ontology at the same time: interleaving the logs to find out the final effect of changes is a difficult task in itself. Therefore, there are a number of ways to represent change that relate V_{old} and V_{new} directly, without taking into account the specific sequence of changes that has actually taken place.

A **structural diff** (Noy and Musen, 2002) provides a map of correspondences between frames in V_{old} and V_{new}. For each from in V_{old} it identifies whether or not there is a corresponding frame in V_{new} (its image) or whether a frame was deleted, or a new frame was added. Figure 4.5 shows a structural diff between ontology versions in Figure 4.3. The structural diff shows that the class Rosé wine was added, the class Riesling was renamed into Weisser Riesling, the class Cabernet blanc changed its superclass, and so on. PROMPTdiff (Noy and Musen, 2002) is an example of a tool that uses heuristics to create a structural diff automatically. It uses persistent identifiers of the frames in different versions, or, if such identifiers are not present, structural relations between ontology elements.

A structural diff provides a *declarative* view of changes: it represents the mapping between versions but it does not contain the operations that one needs to perform to get from one version to another. We call the mapping between frames in different version also the *evolution relation* between frame versions.

A set of **conceptual relations** specifies the conceptual relation between frames *across* versions, i.e. the relation between a frame in V_{old} and the image of that frame in V_{new}. In our example in Figure 4.3, after creating the class Rosé wine, we moved a number of classes that were previously subclasses of White wine to the Rosé wine subtree. In this case, a conceptual change could specify that the class White wine in V_{new} is a *subclass* of the class White wine in V_{old}. Similarly, it could specify that Riesling in V_{old} is *equivalent* to Weisser Riesling in V_{new}. Sometimes, when a consistent interpretation of already annotated datasets is essential, updates are intentionally specified as sets of conceptual

f1	f2	renamed	operation
	Rosé wine	No	Add
Riesling	Weisser Riesling	Yes	Map
Cabernet blanc	Cabernet blanc	No	Map
Vin gris	Vin gris	No	Map
White Zinfandel	White Zinfandel	No	Map
White wine	White wine	No	Map
Wine	Wine	No	Map

level	operation	...	...	reference 1	
Change (direct)	superclass changed			White wine	Rosé wine

Figure 4.5: A table representing a fragment of the structural diff between two ontology versions.

changes. For example, the EMTREE thesaurus,[5] which is used by Elsevier to index scientific publications, specifies updates by defining that specific terms become subsumed by other terms, or that they became synonyms of other terms. In the OntoView system (Klein et al., 2002a) developers can augment a change description with conceptual relations between frames across versions.

The list of change representations in this section is not exhaustive. For example, some systems store concept-history information, associating with each concept a list of concepts that it was derived from, whether a concept was "retired" and which concept replaced it (Oliver et al., 1999). The systems with the primary purpose of data transformation may store a set of operations that is a specific "recipe" for transforming data instances. Other ways to represent change may develop as ontology evolution becomes more and more common.

4.2 Elements of the Framework

Based on the assumptions that we described in the previous section, we can now sketch the main components of our framework for ontology versioning. It consists of:

- a meta-ontology of change operations;
- the notion of complex changes;
- the notion of a transformation set;
- a "template" for the specification of the relation between different ontology versions.

The elements are described in more detail in the next chapters.

4.2.1 Meta Ontology of Change Operations

We defined an *ontology of change operations* that specifies a large number of standard changes to an ontology. This ontology is a central element in our framework because

[5] See `http://www.elsevier.com/locate/emtree`.

different tools using the framework must agree on the basic part of the ontology. As tools use different formalisms for change representation for different tasks or augment information represented in one formalism with information in another, the set of basic operations is the "common language" that they share. This requirement to agree on a common set of basic change operations is similar to the requirement that agents on the Semantic Web share a common ontology language, such as OWL. Defining such an standard set is not unrealistic: once there is a common ontology language (e.g., once OWL becomes a standard), developing and agreeing on an ontology of basic changes is doable. Essentially, an ontology of basic changes is directly related to the ontology language itself and constitutes a set of operations to build an ontology in this language. We present the ontology of change operations in more detail in the Chapter 5.

4.2.2 Complex Change Operations

Besides the basic change operations, the ontology of change operations also contains complex change operations. Complex change operations are operations that are composed of multiple basic operations or that incorporate some additional knowledge about the change.

Complex operations provide a mechanism for grouping a number of basic operations that together constitute a logical entity. For example, a complex operation siblings_move consists of several changes of superclass relations.

Complex changes could also incorporate information about the implication of the operation on the logical model of the ontology. For example, a complex change might specify that the range of a property is *enlarged*, that is, that the filler of the range changed to a superclass of the original filler. To identify such changes, we need to query the logical theory of the ontology. In contrast, basic changes can be specified by using the structure of the ontology only.
Complex operations provide a number of benefits over basic ones.

- Complex operations can be used to improve the user interface for the task of verifying and approving changes. Quite often, an ontology editor performs a number of changes that are all part of one "conceptual" operation. Some complex operations, like sibling_move, capture this knowledge. Visualizing these operations helps the user to verify modifications.
- Complex operations can be used to transform instance data with less data loss. For example, consider the move of a class: if we just had the "remove class" and "add class" operations, we will loose all instances of that class; knowing that the class was moved allows to move the instance data, too.
- Complex operations enables us to determine the effect of operations more precisely. If we only know that the range of a property has changed, we cannot tell anything about the effect on data. However, if we know that the range of the property is *enlarged*, we know that all old instance data is still valid.

The set of complex change operations is never finished or complete. It is always possible to define new complex changes that are useful in some setting. At the same time,

```
changeName oldName=Riesling, newName=Weisser Riesling
addSuperclass child=Cabernet blanc, parent=Rosé wine
addSuperclass child=Vin gris, parent=Rosé wine
removeSuperclass child=Cabernet blanc, parent=White wine
removeSuperclass child=Vin gris, parent=White wine
```

Figure 4.6: A fragment of a transformation set for the example in Figure 4.3

a specific application does not have to use (or commit to) all complex change operations. The set of basic operations is already sufficient to specify all possible transformations. In the Chapter 5 we describe a number of complex operations in more detail.

4.2.3 Transformation Set

Another important element of the framework is a transformation set. It provides a set of change operations that specify how V_{old} can be transformed into V_{new}. The transformation set uses the operations from the ontology of changes. A transformation set is not unique, there are often multiple ways to construct a transformation set for a specific change. The formal definition of a transformation set is:

Definition 1 (Transformation set) *Given two versions of an ontology O, V_{old} and V_{new}, a **transformation set** $T(V_{old}, V_{new})$ is a set of ontology-change operations that applied to V_{old} results in V_{new}. The operations in $T(V_{old}, V_{new})$ can be performed in any order, with one exception: all operations that* create *new classes, properties, and instances are performed first.*

Figure 4.6 presents one possible transformation set for versions in Figure 4.3. The transformation set in the figure contains only basic changes: each change is a single knowledge-base operation. The set can also include complex changes: for example, we can combine two operations that add a superclass and remove a superclass for the same class into a single move operation.
A transformation set is different from a log in the following aspects.

- A log contains a record of *all* the operations that actually took place (including all intermediate steps) during the ontology-editing process. A transformation set specifies only the necessary operations achieve to the resulting change. For example, a log can contain the facts that a concept Riesling was renamed to Rheinriesling, which was again renamed to Weisser Riesling, while a transformation set would only contain the renaming of Riesling to Weisser Riesling (Figures 4.4 and 4.6).
- A log is an *ordered sequence* of actions. A transformation set is an *unordered set* of actions.[6]

[6]There is some very limited partial ordering for transformation sets: mainly, we assume that all "create" operations happen first (see Chapter 5.

- A log is a *unique* representation of the actual change process. A transformation set does not need to be unique. For any two versions of an ontology V_{old} and V_{new} there can be several (and, often, there are many) valid transformation sets. Figure 4.3 shows a fragment of one version of a transformation set. An alternative (and also correct) transformation set that rather than changing the superclasses of the Vin gris class, first deletes the class altogether, and then creates the class with this name again, now as a subclass of the Rosé wine class.
- A log contains operations at a specified, usually low, *level of granularity*. For example, it would usually contain such operations as adding or removing a superclass, but not moving a class or a set of classes from one subtree to another.

Note that if a log of changes between two ontology versions consists of operations that do not undo other operations, this log is by definition a transformation set between these two versions.

A **minimal transformation set** is a special variant of a transformation set. It consists of a set of operations that is sufficient and *necessary* to transform V_{old} into V_{new}.

Definition 2 (Minimal transformation set) *A transformation set $T(V_{old}, V_{new})$ is **minimal** if removing any operation from the set results in a set that is no longer a transformation set from V_{old} to V_{new}.*

The minimal transformation set provides a condensed specification of the change that can be used to re-execute the change and to derive additional information it.

4.2.4 Template for Change Specification

The fourth component of our framework is a template that can be used to describe how two ontology versions are related. The template uses several of the change representations that we distinguished in Section 4.1.3. An assumption behind the template for change specification is that it is often not possible to fill it in completely, because not all information is available. However, we have developed several methods and heuristics that help to derive new information from the available change information. These methods are outlined in Section 4.3.2.

The (partly) filled-in template can be used as a separate mapping ontology that gives information about the consequences of the changes for different types of usage. Chapter 6 in this book shows how the change specification can be used for three different tasks.

The template has the following elements (see Chapter 5 for a more technical description):

Descriptive meta-data: book-keeping information like **date** of release, **author** of the changes, and the number of the versions. This describes the what, when and who of the change and can be used to identify the versions and changes in a setting of collaborative development.

Minimal transformation set: the kernel of the template, as it forms a complete operational specification of the change. It can be used to re-execute the change, to

translate or re-interpret data sets, and as a basis for deriving additional information about the change.

Conceptual relations: the relation between concepts across versions as specified by the ontology engineer. This facilitates data access by improving the interpretation and querying of data sources that were described with different versions of ontologies.

Complex changes: a higher-level description of some of the changes. Together with the minimal transformation set, we can use the complex operations to create data-transformation scripts. Also, complex changes allow us to determine in more detail the effect of changes on data accessibility and specific logical queries. Moreover, visualizing complex rather than basic changes, makes validation and approval tasks much easier for users.

Change rationale: the intention behind the change. The rationale specifies whether the change is a fix of an error, a more specific description, or an update of the real world. The intention can be used to decide which version to use and can help to visualize the change.

For the specification of a rationale, the list suggested by Cimino (1996) could be used as a starting point. Based on an analysis of several medical vocabularies, he distinguishes two reasons of deletion and five reasons for additions. The reasons for deletion are:

- **obsolescence**: deleting a term because it is not in use anymore;
- **discovered redundancy**: deleting a term because the it appears that it is synonym with another term.

The reasons for additions are:

- **simple addition**: adding a term because it represent a truly new concept;
- **refinement**: adding a term to allow greater levels of detail to be specified;
- **precoordination**: adding a term which represents the combined use of multiple existing terms;
- **disambiguation**: adding terms because they distinguish between the different meanings of a homonym term;
- **redundancy**: adding a term because it represents a different name for an existing concept.

Note that this list of Cimino only specifies *possible* reasons, but not a policy for changing vocabularies. This is the reason that “redundancy” exists both as a reason for addition and as a reason for deletion.

4.3 Creating Change Specifications

The previous section described the main static components of the framework for ontology change. We will now sketch the methods to use these components. The methods fall into

two categories: the first category is about methods for finding change information when we just have two version of an ontology (Section 4.3.1), the second category consists of methods for deriving additional change information (Section 4.3.2). In the following chapter, we will describe these methods in more detail. Three tools that implement some of these transformations are described in Chapter 7.

Figure 4.7 gives a graphical overview of the different change representations and some of the possible translations between them.

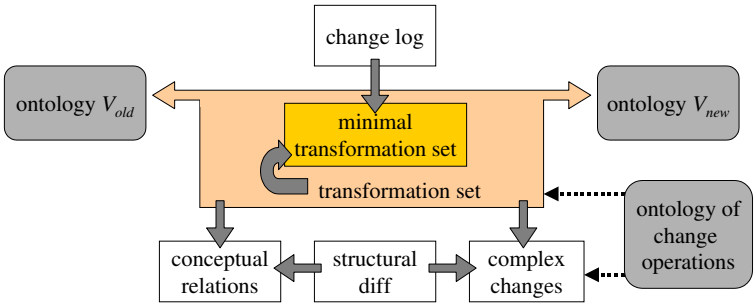


Figure 4.7: A schematic representation of the relations between different change representations: a transformation set between two versions, specified with operations from the ontology of change operations, and possible translations to and from other change representations.

4.3.1 Finding Changes

When we just have V_{old} and V_{new}, we can create a transformation set and a structural diff.

V_{old} and V_{new} → transformation set If we have both V_{old} and V_{new}, we can compare the two versions to create automatically a transformation set between them. Two tools that help in comparing ontology versions and creating a transformation set are described in Chapter 7.

V_{old} and V_{new} → structural diff Starting from V_{old} and V_{new}, we can also create a structural diff between both. This specification maps the concepts and properties in the old version to their counterpart in the new version. If concepts in an ontology have immutable concept ids, a simple tool can create a diff between versions identifying for each frame F in V_{old} its image in V_{new}. If we do not have immutable concept ids, tools such as PROMPTdiff (Noy and Musen, 2002) use a set of heuristics based on concept names, class-tree structure, and concept definitions to create a structural diff.

4.3.2 Deriving New Information

We can use change information that we already have (for example, the transformation set or structural diff that is created in the previous section) to derive additional information about the change. As a result, having *some* information about change enables us to complete the change specification by deriving additional elements of it. Even in the case that we cannot fill in all the pieces, we might still be able to support tasks that we could not support before. We now describe some of the transformations from one change description to another.

Change log → minimal transformation set Many ontology-editing tools provide logs of changes (e.g., Protégé, OntoEdit). These changes are often at the level of simple knowledge-based operations: adding a superclass to a class or removing one. We can transform logs into transformation sets by translating the operations into our vocabulary of basic changes and finding and removing all redundant changes.

Transformation set → minimal transformation set Similarly, we can transform transformation set into minimal transformation sets by finding and removing all redundant changes.

Transformation set → complex changes If we have a transformation set consisting of basic operations, we can use heuristics to combine these basic operations to create complex change operations. For instance, if we have a set of siblings in a class hierarchy and each of these siblings had the same class added as a superclass and the original superclass removed, we can infer that the whole set of siblings was moved together from one part of the hierarchy to another. In addition to the set of basic changes, we may need direct access to V_{old} to find complex changes.

Structural diff → complex changes If we have a structural diff, we can use the information in it to create more useful change descriptions. Consider for example the structural diff in Figure 4.5. Knowing that Riesling became Weisser Riesling and that the name is the only difference between the two classes, we can identify a changeName operation.

Transformation set → conceptual relations If we have a transformation set with both basic and complex operations defined between versions, we can use a set of heuristics to suggest conceptual relations between frames in versions to the user. For example, if we add a property to a class, we might suggest to the user that the new version of the class has become a subclass of the old version.

Structural diff → conceptual relations Similarly, we can use a structural diff to derive conceptual relations. For instance, the mappings of a structural diff directly suggest equivalence relations betweens concepts.

V_{old} V_{new} **and structural diff** $\rightarrow$ **conceptual relations** When both versions of the ontology are available and we have the evolution relation between concepts, we can use a automated reasoner to derive conceptual relations between the concepts in the old and new version of the ontology. Of course, this only works if a decidable logic underlies the ontology language. Tot do so, we add the old concept definition to the new version of the ontology and compute the subsumption relation between the old and new version of the concept.

4.4 Summary

In this chapter, we described the basic assumptions and elements of our framework for ontology evolution. The main elements of the framework are the ontology of change operations, the notion of complex changes, the transformation set, and the template for change specification.

The methods that we described illustrate how we can derive new change information from other change representations. This makes it possible to supplement information about change, and therefore to support ontology-evolution tasks that the original change information did not allow.

In the next chapter, we describe the ontology of change operations and the complex changes. The succeeding chapter shows how the change specification can be used for different ontology tasks. Tools that implement some of the methods and that support these tasks are described in Chapter 7.

Chapter 5

Ontology of Change Operations

In the previous chapter, we have outlined our framework for ontology evolution. A key element of the framework is the *transformation set*, which represents a complete operational specification of the change. A transformation set is specified as a number of change operations. In this chapter we define the vocabulary for these change operations and organize them into an *ontology of change operations*.

In addition, we define a *change specification language*, based on the ontology of change operations. We do this by proposing an RDF-based syntax for the operations and linking the ontology of change operations to the meta-model of the ontology.

The chapter is organized as follows. We start with a discussion of the objectives and requirements for a change specification language. Then, we use the metamodel of two well-known ontology representation formalisms, i.e. OWL and OKBC, to define a set of basic change operations. This is followed by a discussion of a number of complex changes in Section 5.4. In Section 5.5 we extend the ontology of change operations for OWL to a general change specification language.

5.1 Usage and requirements

The general goal of a change specification language is to provide a vocabulary and syntax to express an accurate specification of a change. Apart from its specific use in our ontology evolution framework, a change specification language can be useful for other tasks as well. In the following scenarios a change specification could help.

Change reverting or re-execution A description of the changes that are performed in an ontology can be used to undo them, thus bringing the ontology back to its original state. Also, the specification can be used to re-execute the changes in a variant of the ontology, or on data sets that are described with the ontology.

Effect Analysis A complete change specification can be used as an important source for analyzing the effect of the changes on the compatibility of the ontology with

data sets (Heflin and Hendler, 2000) or with other ontologies (Stuckenschmidt and Klein, 2003).

Tool Interaction Tools need to exchange unambiguous specifications of modifications when they interact in an ontology development process. If ontologies are developed in a collaborative process (Pinto and Martins, 2002; McGuinness, 2000), specifications of the changes performed by one developer need to be communicated to other developers and/or applications.

The transaction logs that are produced by some ontology editors, e.g. OntoEdit (Sure et al., 2002) and Protégé [1] are examples of change specifications. However, these logs have proprietary formats, while for a useful interaction between different tools a *standard* language is needed. The language that we present in this chapter (in Section 5.5) aims to be such a language. To function as a standard, it should fulfil a number of requirements.

Sufficient expressiveness: The language should be sufficiently rich to specify all possible modifications to an ontology, i.e. it should be capable of representing arbitrary transformation sets between ontologies.

Because a transformation set specifies changes in the *specification* of an ontology, the language should be complete with respect specification changes. This implies that it should be capable to express changes that do not constitute a logical change, but it does not require the language to be capable of specifying *representation* changes (see Section 4.1.1 for an explanation of the different levels of change).

Minimality: The required commitment of users should be minimal, i.e. the users should not be forced to commit to a larger vocabulary than strictly necessary. This requirement follows from the distributed context of the framework for ontology evolution: different tools and / or users might need to exchange information about ontology change, and there should be no unnecessary obstacles for the adoption of the language.

Different levels of granularity: A general language for ontology change should support different levels of granularity for change descriptions, as different tasks require specifications at different levels (see also Chapter 3). For undo-operations, a fine-grained specification is needed, whereas effect analysis often requires a specification at a more coarse-grained level.

These requirements are in some sense conflicting and we have to find a balance between them for our change specification language.

From the many possible change representations (see Chapter 4), we have chosen to base our change representation on **change operations**. Change operations are precisely defined additions, removals or modifications to the definition of a concept, a property, or an ontology as a whole. If we relate this to the different types of ontology changes that are described in Section 4.1.1, it means that change operations modify the *specification* but are ignorant about the *representation*. As a consequence, according to the above

[1] The Protégé project, `http://protege.stanford.edu`.

definition there exist no operation that specifies a change to an equivalent representation. Our idea of change operations is comparable with the taxonomy of operations for ER-model changes that is proposed by Roddick et al. (1994).

The main reason for supporting change operations as change representation is that they can be used for a *complete* specification of transformations. Using the change operations, one can specify a recipe that is sufficient to transform one version of an ontology into another version of the ontology. However, a set of change operations does not give a complete specification of the *change* itself, it only gives a complete specification of the operational transformation. There are many aspects of an ontology change, such as the reason, the conceptual consequence and the meta-data that are not covered by change operations.

A second reason for the choice for change operations is the third requirement, i.e. allowing different levels of granularity. Change operations can be aggregated into composite operations that perform several modifications in one step. By choosing for either composite operations or simple operations, one can vary the level of granularity of a change specification.

To produce the list of basic change operations on ontologies, we exploit the meta-model of the ontology language. Each ontology that is represented in a specific ontology language is an instantiation of the meta model of that language. Therefore, each change in an ontology can be seen as a set of additions, removals and modifications of one of the elements of the meta model.

Using the meta-model as a source for the change operations has two advantages. First, we abstract from representational issues, as the meta model describes the modeling constructs and not the representation. In contrast, if we would look at the syntax of the language, we would define changes at the representation level. Second, a by iterating over all elements of the meta model of an ontology language, we get a list of operations that is *complete* with respect to the possible changes of ontologies. We can specify every possible change with the “add” and “delete” operations for each element of the knowledge model,

In the next two sections we look at two different ontology languages. First, we look at the OKBC knowledge model (Chaudhri et al., 1998a). Second, we will consider the Web Ontology Language OWL (Bechhofer et al., 2004), a W3C recommendation. By comparing their respective knowledge models, we will conclude that OWL is a superset of OKBC in most aspects. This suggests that the procedure that is followed to define change operations for OWL can be applied to other languages as well.

5.2 OKBC Ontology Language

The Open Knowledge Base Connectivity is a protocol for accessing knowledge bases. The OKBC *knowledge model* is the implicit representation formalism that underlies all the operations provided by OKBC. It supports an object-oriented representation of knowledge. The knowledge model is extensive and well defined, and has several implementations (Fergerson et al., 2000; Farquhar et al., 1997). In this section we give an informal description of the OKBC knowledge model. For a formal characterization, we

refer to chapter 2 in the OKBC specification (Chaudhri et al., 1998b).

5.2.1 OKBC Knowledge Model

As OKBC is a frame-based representation mechanism, a *frame* is the central object in the model. A frame represents an entity in the domain of discourse. There are three main types of frames: *class frames*, which represent sets of entities, *slot frames*, which represent binary relations, and *individual frames*, which represent single entities.

If entities are member of a class, they are said to be *instance_of* that class. The other way around, the class is called the *type_of* that instance. A class can be a *subclass* of another class: if this is the case, then all instances of the subclass are also instances of the other class.

All frames can be related via slots to other frames or constants. Slots that are associated with a frame are called *own slots* of such frame. For example, an individual frame "George_Bush" can be related with the own slot "president_of" to the frame "United_States". Class frames can also have own slots, although this is much less common. An example is the identifier that is associated with classes in some categorization systems: e.g. in UNSPSC the class of "ball_point_pens" is associated via the own slot "code" to the string "44.12.17.4".

Besides own slots, class frames can also be associated with a collection of *template slots*, that describe slots that are considered to hold for all members of that class. For example, the class "blue_ball_point_pens" can have the template slot "ink_color" to the constant "blue", meaning that every *instance* of "blue_ball_point_pens" should have the value "blue" for the slot "ink_color". Template slots of a class inherit to its subclasses.

Slots that are related to a frame can have associated with them a set of *facets* and *facet values*. Facets and facet values describe characteristics of the combination of a frame and a slot. For example, the facet "value-type" and value "President" associated with the slot "rules" in the class frame "Republic" specifies that that value of the slot "rules" for each instance of a republic should be an instance of a president (as opposed to monarchies, where the slot "rules" should relate to an instance of a king or queen).

The OKBC knowledge model contains a number of standard facets, concerning the *value-type*, the *inverse* relation, the *cardinality* and the *equivalence* of the slot among others.

Figure 5.1 shows a UML class diagram of the main elements of the OKBC knowledge model. For the sake of clarity, we made a few simplifications. First, we modeled constraints as a separate class, although constraints are either specified by a facet or are defined as global constraints on a slot. In the OKBC model, all constraints exist in two different variants: both as a facet and as an slot on a slot. As a second simplification, we didn't show all types of constraints. The constraints that are missing in the picture are:

- disjointness of slots;
- numeric minimum and numeric maximum for slot values;
- subset of values of a slot;
- collection type of multiple slot values: multiple values are either treated as *set*, *list* or *bag*.

5.2.2 Change operations for OKBC

The OKBC meta model gives us some insight in the main elements of an ontology change language. In principle, a simple change of an ontology is either an addition, removal or modification of one of the elements of the meta model. The cardinality constraints in the meta-model function as additional restrictions on the list of possible change operations. For example, the 'inverse slot' constraint requires the specification of the name of opposite slot, which is represented by the cardinality "1" on the association between "InverseC" and "Slot" classes in the meta-model. Therefore, it is not possible to add or remove an inverse constraint on itself, but it should also coincide with the addition or removal of a value for the inverse constraint (i.e. a slot).

As an example, we will list a number of change operations for the OKBC model. For this, we use the Slot class and its direct relations in the OKBC meta model. Each of these elements can be added and removed, and if they have an argument value, they can be modified. Note that abstract classes in the meta model can not be instantiated and therefore they are not used to generate change operations. This procedure results in the operations that are listed in Table 5.1.

Operation	**Description**
Add_Slot	create a slot
Remove_Slot	delete a slot
Add_Domain	add a domain attribute to an existing slot, together with an argument to the attribute (i.e. a class name)
Remove_Domain	remove a domain attribute to an existing slot
Modify_Domain	change the reference from one class to another class
Add_Some-Value_Constraint	add a global existential constraint to the slot, together with a reference to an individual
Remove_Some-Value_Constraint	remove the global existential constraint
Modify_Some-Value_Constraint	change the reference from one individual to another individual
Add_Cardinality_Lower_Bound	add a global minimal cardinality constraint to the slot, together with a value (a non-negative integer)
Remove_Cardinality_Lower_Bound	remove a global minimal cardinality constraint to the slot
Modify_Cardinality_Lower_Bound	change the value for the minimal cardinality constraint
...	similar operations for the other cardinality constraint and the inverse and equivalence constraint

Table 5.1: A number of change operations for ontologies expressed in an OKBC knowledge model.

5.3 Web Ontology Language OWL

We will now turn our attention to the Web Ontology Language OWL. We will follow the same procedure as with OKBC: we first present the meta model, and then we will define a list with change operations.

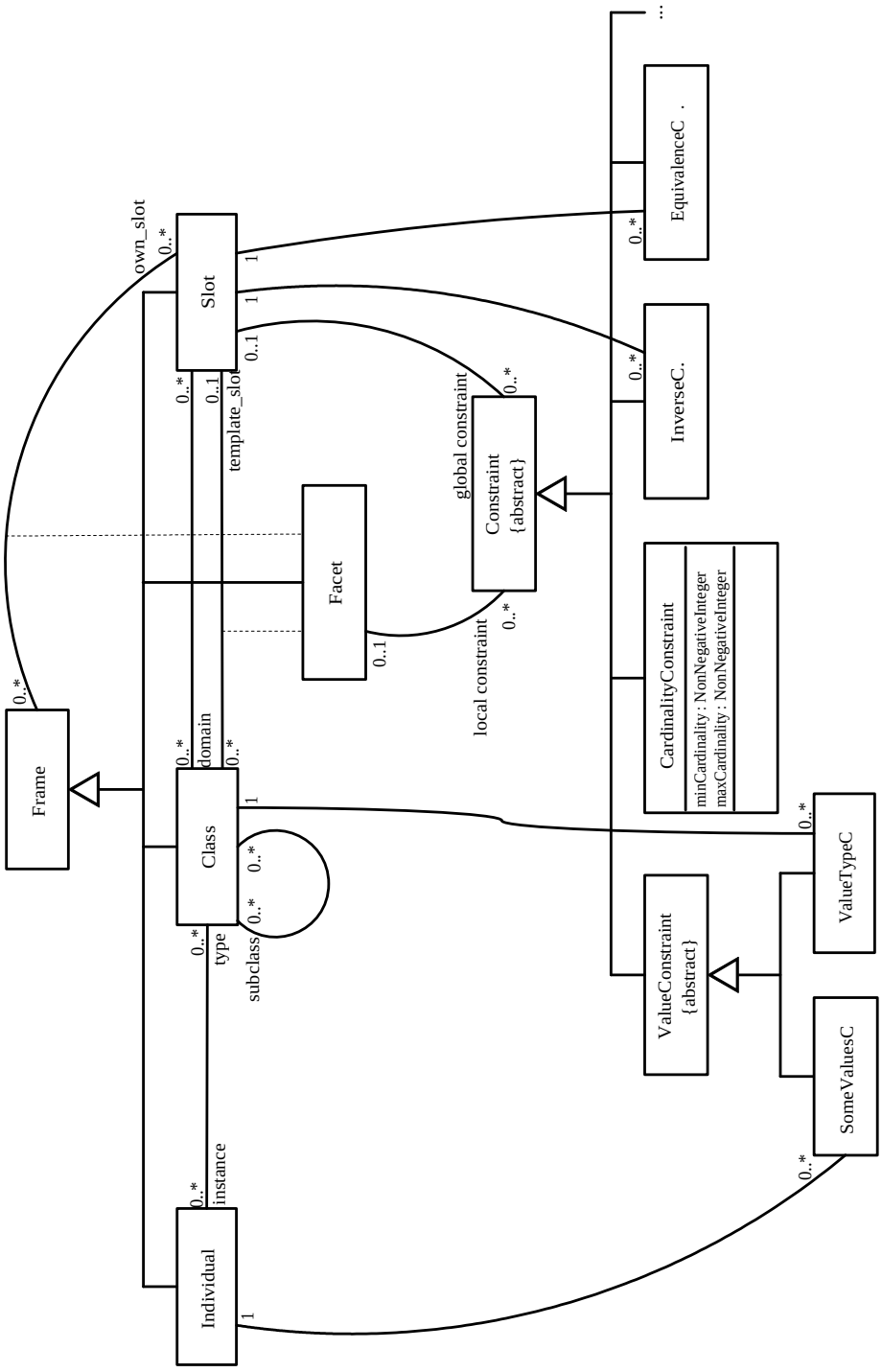


Figure 5.1: A simplified representation of the OKBC knowledge model.

5.3.1 OWL meta model

OWL comes in three different flavors: OWL Lite, OWL DL and OWL Full. OWL Lite is the simplest variant. Its language constructs are a subset of the language constructs in OWL DL. Also, in OWL Lite references to classes are often restricted to *named* classes, while in OWL DL anonymous class definitions can be used. For example, in OWL DL it is possible to state that a class is equivalent to the intersection of two other classes, while this is not possible in OWL Lite.

The difference between OWL DL and OWL Full is more subtle. They share the same vocabulary and modeling constructs, but OWL DL places several constraints on the use of the language constructs. According to the OWL Language Reference (Bechhofer et al., 2004), the main constraints are the following.

- "OWL DL requires a pairwise separation between classes, datatypes, datatype properties, object properties, annotation properties, ontology properties (i.e., the import and versioning stuff), individuals, data values and the built-in vocabulary. This means that, for example, a class cannot be at the same time an individual.
- In OWL DL the set of object properties and datatype properties are disjoint. This implies that the following four property characteristics:
  - inverse of,
  - inverse functional,
  - symmetric, and
  - transitive

  can never be specified for datatype properties.
- OWL DL requires that no cardinality constraints (local nor global) can be placed on transitive properties or their inverses or any of their superproperties.
- Annotations are only allowed if they are explicitly typed as AnnotationProperty's and if they are not used in property axioms.
- All axioms must be well-formed, with no missing or extra components, and must form a tree-like structure. This implies that all classes and properties that one refers to should be explicitly typed as OWL classes or properties, respectively.
- Axioms (facts) about individual equality and difference must be about named individuals."

In the remainder of this chapter, we will focus on OWL Full, as the other variants can be seen as restricted versions of the full language. Where necessary we will make the distinction between the variants.

OWL differs from OKBC in a number of aspects. Besides a few terminological differences (the most notable is that a slot is called a property), we can see the following differences.

- In OWL, the set of slot constraints is divided into global and local constraints. That is, a specific constraints is either a global constraint that holds for all values of the slot (e.g. "functional"), or it is a local constraint that only restricts the values of

a slot when used in a specific class (e.g. "has-value"). In OKBC, all constraints can be applied both globally and locally. The following constraints can be applied *locally* in OKBC, but only globally in OWL:

  - inverse of a slot;
  - equivalence of a slot;
  - values are subset of the values of another slot (called "subslot" if applied at global level).

- There is a difference between the built-in constraints in OKBC and OWL. The following constraints in OWL do not exist in OKBC:

  - symmetry of slots (global);
  - transitivity of slots (global);
  - "inverse functional" constraint[2] (global);
  - existential constraint (local).

  On the other hand, the following built-in constraints in OKBC are missing in OWL:

  - disjointness of a slot;
  - numeric minimum and maximum for values of a slot;[3]
  - collection type of slot values.

  The difference between OWL and OKBC with respect to slot constraints is summarized in Table 5.2.
- In OWL "slot–facet–value" triples are classes themselves. That is, a constraint on a slot, called a "property restriction", defines a class. For example, the property restriction "age–hasvalue–27" defines the class of things that have the value "27" for the slot "age", i.e. the class of all 27 years old things.
- In OWL arbitrary classes can be formed by combining other classes via boolean operations, forming so-called *complex classes*. In OKBC, it is only possible to use the *union* of classes as filler for the value-type restriction.
- Slots in OWL are divided into slots that can have instances as their value, and slots that can have data type values.
- Classes and individuals in OWL can be declared as equivalent or disjoint.

Figure 5.2 shows a UML class diagram of the OWL meta model.

Based on these differences, we can not conclude that one knowledge model is contained in the other model. However, when we look carefully at the differences, we can see that the elements of OKBC that are missing in OWL are quite rare. For example, it is difficult to think of examples or a practical usage of the local equivalence constraints on slots, or a local inverse constraint. It is likely that these constructs are present in OKBC for reasons of symmetry with the global constraints. The disjointness of slots seems to be

[2]This can be represented in OKBC by defining a global cardinality restriction on the inverse of a slot.

[3]Although this can be mimicked in OWL by defining a data type that is restricted to a certain range of numeric values.

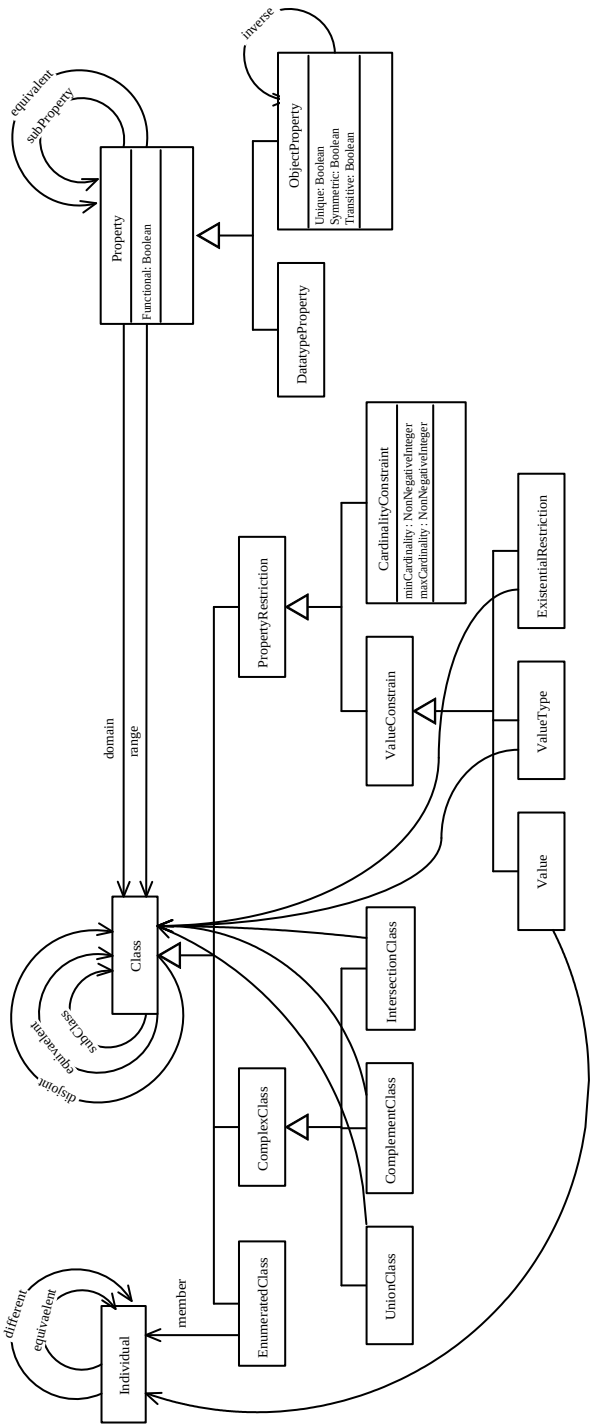


Figure 5.2: A UML representation of the OWL meta model.

constraint	OWL	OKBC
symmetry	G	-
transitivity	G	-
inverse functional	G	-
inverse	G	GL
equivalence	G	GL
values are subset	G	GL
disjointness	-	GL
numeric bounds	-	GL
collection type	-	GL
existentiality	L	-

Table 5.2: The slot constraints for which OKBC and OWL differ. A 'G' means that the constraint can be specified at global level, a 'L' means applicability at the local level, and a minus sign (-) means not applicable at all.

the most useful construct that is missing in OWL. Besides these aspects, we can consider the OWL knowledge model as almost a superset of OKBC for practical applications. In the remainder of this book we will use the OWL (Full) knowledge model.

5.3.2 OWL Change Operations

Based on the meta-model that we sketched above, we can produce a complete list of change operations for the OWL variants.

The main effect of the differences between the variants of OWL on the change operations is that the operations for OWL Lite are a subset of the operations for OWL DL and OWL Full. In addition, the values for the arguments of some of the operations in OWL Lite are restricted. For example, in OWL Lite, the argument of the add_class_equivalence operation is restricted to a class ID in OWL Lite, but can be an arbitrary class definition in OWL DL and OWL Full.

The differences between OWL Full and OWL DL do not have direct effect on the set of change operations. The reason is, that for each of the constraints multiple axioms have to be checked to find a violation of the constraints. For example, to find an illegal inverse-functional datatype property X, one needs the information that (1) X is a datatype property, and (2) X is inverse-functional. Change operations, however, operate on named entities and do not have any other information about the context of the operation, i.e. about other axioms in the model. As a result, it is not possible to enforce the OWL DL constraints by restricting the set of operations.

Table 5.3 gives a summary of the change operations for the OWL language. The complete list of operations is printed in Appendix B and are defined in the ontology at `http://ontoview.org/changes/2/1`.

For most of the operations, defining them was very straightforward, as they are just additions, deletions and modification of elements of the knowledge model. However, there are some additional considerations behind the definitions, which are discussed below.

Object	Operation(s)	Argument(s)
class related operations		
class	add, remove	class definition
restriction	add, remove	restriction definition + type
restriction filler	modify	2 restriction fillers
cardinality upper/lowerbound	add, remove	cardinality restriction
cardinality upper/lowerbound	modify	property ID + 2 values
restriction type	make $\forall$, make $\exists$	restriction definition
superclass relation	add, remove, modify	1 / 2 class definitions
class disjointness	add, remove, modify	1 / 2 class definitions
class equivalence	add, remove, modify	1 / 2 class definitions
property related operations		
property	add, remove	property definition
domain	add, remove, modify	1 / 2 class definitions
range	add, remove, modify	1 / 2 class definitions
superproperty relation	add, remove, modify	1 / 2 property definitions
property equivalence	add, remove, modify	1 / 2 property definitions
property inverse	add, remove, modify	1 / 2 property definitions
property symmetry	set, unset	property ID
property functionality	set, unset	property ID
property transitivity	set, unset	property ID
property inverse-functionality	set, unset	property ID
property type	make datatype / make object	property ID
individual related operations		
individual	add, remove	individual definition
individual equivalence	add, remove	individual definition
individual distinctness	add, remove	individual definition
operations for both classes, properties and individuals		
resource type	add, remove, modify	1 / 2 class ID's
resource label	add, remove, modify	1 / 2 strings
resource comment	add, remove, modify	1 / 2 strings
resource annotation	add, remove, modify	property ID + 1 / 2 values

Table 5.3: Overview of the change operations for the OWL languages.

Modify versus add / delete: The list of change operations also contains "modify" operations, which specify that an old value is replaced by a new value. For example, a Modify_Range operation specifies that the filler of the range of a property has changed. Of course, these operations can also be formed by combing a "delete" and an "add" operation. We have included them in the list of basic changes because this information is often available. For instance, logs of changes provided by tools will often contain information on modifications.

Arguments: All "modify" operations take two arguments, i.e. both the old value and the new value. In principle, this is not required for the execution of the operations. However, one of the goals of a transformation set is to have a *complete* specification that allows reversing the changes. To achieve this goal we also need the old value for the modification operations.

Property restrictions vs. subclass / equivalence relations: From a frame-based perspective, there is a difference between attaching slots to classes and adding subclass relations between classes. However, from a Description Logic point of view, adding property restrictions to classes is exactly the same as adding a subclass relation between the restriction and the class (or adding an equivalence relation in case of "defined classes"). Because it is based on Description Logics, in OWL adding property restrictions is also the same as adding subclass / equivalence relations.

Although the logical meaning is the same, the idea behind the action is in most cases quite different. We therefore decided to introduce different operations for adding a subclass relation between two classes and adding a subclass relation between a class and a property restriction (and similar for the equivalence relation).[4] To distinguish between property restrictions that represent *necessary* conditions (resulting in a subclass relation between the class and the restriction) the and those that represent *necessary and sufficient* conditions (resulting in an equivalence relation), we give an extra argument to the Add_Property_Restriction operation.

Cardinality: For the operations that modify the cardinality we make a distinction between upper bound modification and lower bound modifications. This causes some inefficiency (e.g. a change in the exact cardinality requires two operations), but it makes the specification complete and gives a fine-grained granularity. Also, it reflects the way in which cardinalities are represented in the meta model better.

Changing complex arguments: Many operations can be used with complex arguments, for example boolean class definitions or anonymous class descriptions. The list of operations do not provide support to alter these complex definitions itself. If we would do this, the meaning of the operations would become very little, as they would come close to syntactic changes of definitions. Instead, we use complete (anonymous) definitions as arguments to the operations. This means that we only allow 'top-level' modifications, i.e. adding and removing restrictions and complex classes as a whole, but no arbitrary modifications.

5.4 Complex Operations

The operations that we have defined in the previous sections are called **basic change operations**. Each of the basic change operations modifies only *one* specific feature of the OWL knowledge model. This basically means that there are "add" and "delete" operations for each element of the knowledge model. This set of operations ensures the completeness of the ontology of basic changes since it is sufficient for defining a transformation set from any ontology version V_{old} to any other ontology version V_{new}. While not being the most useful or efficient, such transformation set can contain the operations that delete all elements in V_{old} and then add all elements in V_{new}.

[4] We can, however, not *enforce* that all restriction additions are represented using this operations. It is very well possible to represent it using a regular Add_Subclass operation.

In this section, we extend the set of operations with **complex change operations**. In Chapter 4, we already introduced these operations and explained why they are useful. We now describe them in detail.

5.4.1 Types of Complex Operations

There are two dimensions that can be used to distinguish between different types of complex operations. On one hand, there is a distinction between *atomic* and *composite* operations, on the other hand there is a distinction between *simple* and *rich* operations. Figure 5.3 shows these two dimensions of complex operations.

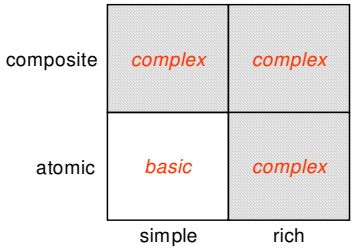


Figure 5.3: The relation between atomic–composite and simple–rich operations.

Composite operations provide a mechanism for grouping a number of basic operations that together constitute a logical entity. For example, a complex operation siblings_move consists of several changes of superclass relations. **Atomic operations** are operations that cannot be subdivided into smaller operations.

Rich changes are changes that incorporate information about the implication of the operation on the logical model of the ontology. For example, a rich change might specify that the range of a property is *enlarged*, that is, that the filler of the range changed to a superclass of the original filler. To identify such changes, we need to query the logical theory of the ontology. In contrast, **simple changes** can be detected by analyzing the structure only.

There are several procedures that can be followed to define complex operations. We explain them and refer to Table 5.4 for examples. To define *rich* operations, we can first extend all basic operations that "modify" a class filler (e.g. Modify_Range, or Modify_Subclass) with a proposition that specifies the subsumption relation between the old and the new filler. Thus doing, we can define two variants of each modify operations that are called "changed to superclass" and "changed to subclass". This is the first block of changes in Table 5.4. Second, we can do something similar for operations that have properties as arguments, defining "changed to subproperty" and "changed to superproperty" variant. Similarly, we can incorporate knowledge about the "direction" in which the cardinality is changed, i.e. whether it increased or decreased. This is the reason for having operations like Increase_Cardinality_Upperbound as well as operations like Restrict_Cardinality (see third block in table). Other methods that incorporate other knowledge might be possible as well.

To define *composite* operations, we have to find useful or common combinations of atomic operations. The usability of a composite operation depends on the specific goals in an ontology evolution situation and can therefore not be determined beforehand. Specific goals may come with their own set of useful complex operations. To find some of the common operations, we used the analysis presented in Chapter 3, where we looked at current change management strategies. This resulted in a list of common "hierarchy operations", like Add_Subtree. These operations are listed in the second block of the right column in Table 5.4.

Some combinations of basic operations are common because they represent a "mental" operation. This is the case for the operations Change_To_Primitive and Change_To_Defined, which changes the relation between a class and its property restrictions to "subclass" relations or "equivalence" relations, respectively.

Another reason for defining a composite operation can be the fact that there is a syntactic shortcut in the language. A shortcut states multiple axioms with only one language construct. If such a shortcut ia used, the axioms that are defined by it are performed in combination. An example of this is the `allDifferent` statement in OWL: it states that all mentioned individuals are different from each other, resulting in pairwise disjointness axioms between all individuals. To support this, we defined the operations Add_Disjoint_Set and Remove_Disjoint_Set, which both take the whole set of individuals as argument. Note that does not make sense to define operations that adds or removes members from the set of disjoint individuals, as we cannot identify the set other than by its complete extension.

It should be clear from the above that the list of operations in Table 5.4 is not complete or finished, as new complex operations can always be defined. As is sketched in the framework in the previous chapter, in some situations it can be useful to distill complex operations from a set of basic operations. For this process, called *enrichment* we use rules and heuristics that are described in Chapter 6.

5.4.2 Hierarchical Ordering of Operations

Above we listed a number of complex operations for the OWL language, and in Section 5.3.2, we presented the set of basic operations. Especially for rich operations, it is often the case that the complex operations are a specific variant of the basic operation. For example, increasing the cardinality of a property restriction is a kind of modification to a cardinality restriction. Therefore, it makes sense to build a hierarchy of all operations.

We have done this by modeling each change operations as a class, and defining subsumption relations between these classes. For example, the basic change Modify_Domain has two (complex) subclasses: Modify_Domain_To_Superclass and Modify_Domain_To_Subclass. Thus doing, we have created an ontology of operations. In this ontology the characteristics of each change type are specified via property restrictions.

By organizing the change operations in a class hierarchy we exploit the inheritance mechanism to specify common properties of change operations in an efficient way. For example, all changes in property restrictions require two arguments to identify the source, namely the class identifier and the property identifier. Since all changes in property

Operation	Type
Restrict_Range	R
Modify_Range_To_Subclass	R
Extend_Range	R
Modify_Range_To_Superclass	R
Restrict_Domain	R
Modify_Domain_To_Subclass	R
Extend_Domain	R
Modify_Domain_To_Superclass	R
Modify_Equivalence_To_Subclass	R
Modify_Equivalence_To_Superclass	R
Modify_Disjointness_To_Subclass	R
Modify_Disjointness_To_Superclass	R
Modify_Type_To_Subclass	R
Modify_Type_To_Superclass	R
Modify_Superclass_To_Subclass	R
Modify_Superclass_To_Superclass	R
Modify_Restriction-Filler_To_Subclass	R
Modify_Restriction-Filler_To_Superclass	R
Modify_Equivalence_To_Subproperty	R
Modify_Equivalence_To_Superproperty	R
Modify_Inverse_To_Subproperty	R
Modify_Inverse_To_Superproperty	R
continued in next column	

continued	
Operation	**Type**
Restrict_Cardinality	CR
Extend_Cardinality	CR
Increase_Lowerbound	R
Decrease_Lowerbound	R
Increase_Upperbound	R
Decrease_Upperbound	R
Add_Subtree	C
Delete_Subtree	C
Move_Subtree	C
Delete_Subclasses	C
Delete_Class_And_Move_Siblings_Up	C
Move_Siblings	C
Move_Siblings_Down	CR
Move_Siblings_Up	CR
Move_Siblings_To_New_Subclass	CR
Move_Slots_To_New_Referring_Class	C
Split_Into_Multiple_Siblings	C
Merge_Multiple_Siblings	C
Change_To_Primitive	CR
Change_To_Defined	CR
Add_Disjoint_Set	C
Remove_Disjoint_Set	C

Table 5.4: A number of complex change operations for OWL

restrictions are subclasses of Property_Restriction_Change, we can easily specify this fact for all operations at once.

Some of the operations in the change ontology are abstract. These operations cannot be instantiated, they are only introduced for structuring purposes. This is represented in the ontology by annotating them with a property "role", which is either "concrete" or "abstract". For convenience, we have called the abstract categories ..._Change, while operations that can be instantiated are called are Add_..., Remove_... and Modify_..., etc. The complete ontology of change operations can be found at `http://ontoview.org/changes/2/1`.

5.5 Ontology Change Language

An hierarchy of change operations is not yet a language that can be used to specify changes. In this section, we explain how we build a change specification language from the hierarchy of change operations. This language provides the constructs and the structure for specifying ontology change.

5.5.1 Model of Ontology Change

The language for ontology change is defined by extending the hierarchy of change operations into an "ontology about ontology change". In this ontology we model the relations between the most important concepts around ontology change. With this ontology, we can specify an actual change as instance data that conforms to the ontology. Figure 5.4 shows a graphical representation of the model.

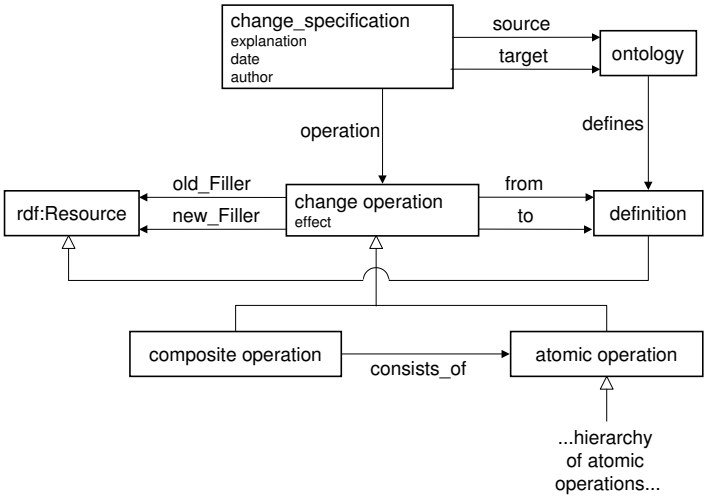


Figure 5.4: The ontology of ontology change.

The most general concept in the hierarchy of operations is called "change operation". This concept has two subclasses. One is a "composite operation", which is either one of the predefined composite operations that are described in Section 5.4, or an arbitrary combination of atomic operations. The latter type of composite operations can be formed by relating several atomic operations via the consists_of relation to a composite operation. These arbitrary composite operations provide a mechanism to cluster related operations and specify their shared characteristics. We assume that all properties of a composite operation inherit to its components. It is in some sense comparable with a transaction mechanism in databases. The other subclass of a "change operation" is an "atomic operation". This is the top-level concept of the hierarchy of change operations that is defined in the previous sections.

This "change operation" concept is related to actual definitions of concepts via a "from" and a "to" property, which refer to the old and new version of a definition, respectively. "Definitions" are the subset of resources that define classes, properties or individuals.[5] A set of change operations is called a "change specification". A change

[5]This definition is intentionally informal. A class or property definition is nothing more than a specification of a number of properties for a named resource, and can thus only be identified as a resource. In practice, definitions are formed by all statements in one file about a named resource that use properties from the OWL

specification has a source and target ontology, which define concepts and properties. The "defines" relation between an ontology and definitions is implicitly derived from the file containment, i.e., we assume a "defines" relation between the ontology resource and the definition if the definition is contained in an ontology.

All change classes use the "from" and "to" properties to refer to the source and target of the change, respectively.[6] In addition, most change classes have properties that specify an argument for a change operations. For example, one of the arguments for the operation Modify_Upperbound that changes the maximum cardinality is an integer specifying the new cardinality restriction.

In our ontology of change, we have made all operations instances of a "Change_Class". This class has property "effect", which allows us to annotate the *class of* operations (i.e. not a specific change in one ontology, but the operation in general) with the effect of the change. In Chapter 6 we discuss in more detail what the different types of effects for changes can be. By defining several subproperties of "effect" we can distinguish between the different tasks that are influenced by a change.

5.5.2 Syntax and Interpretation of Change Specification

Since we defined an ontology to structure the operations and model ontology change, we do not need to define a syntax for the representation of actual changes. Instead, we can use the representation format that comes with the ontology language that we use. Our ontology of basic change operations uses OWL as "ontology language". This implies that actual changes can be represented as RDF data. In its simplest form, a change description looks as follows.

```
<ov:Set_Transtivity>
  <ov:from rdf:resource="..ontology/1/#larger"/>
  <ov:to rdf:resource="..ontology/2/#larger"/>
</ov:Set_Transtivity>
```

This piece of RDF data specifies that there is a change of type Set_Transtivity from the resource "larger" in one version of an ontology to the resource "larger" in another version of the ontology. In other words, the property "larger" is declared as transitive.

For operations that take datatype values as arguments, we use the datatype properties "old_Value" and "new_Value" instead of the "old_Filler" and "new_Filler" properties. The following example represents that a minimum cardinality restriction of 4 on the property "hasWheels" is added in the class "Car". The ontology of changes provides the properties "onProperty" and "value" to specify the necessary arguments.

```
<ov:Add_Lowerbound>
  <ov:from rdf:resource="..ontology/1/#Car"/>
  <ov:to rdf:resource="..ontology/2/#Car"/>
  <ov:on_Property rdf:resource="..ontology/1/#hasWheels"/>
  <ov:new_Value>4</ov:new_Value>
</ov:Add_Lowerbound>
```

language specification.

[6] Additions and removals of classes or properties are the only exceptions, since they only have one of these two properties.

The fact that both our ontology of changes and OWL itself can be represented in RDF, makes it possible to represent complex arguments. The RDF representation of the complex argument can just be inserted as value of a property in our ontology. For example, to express that a person became a vegetarian, the following RDF definition can be used.

```
<ov:Modify_Restriction_Filler>
  <ov:from rdf:resource="..ontology/1/#Person"/>
  <ov:to rdf:resource="..ontology/1/#Person"/>
  <ov:onProperty rdf:resource="..ontology/1/#eats"/>
  <ov:old_Filler rdf:resource="..ontology/1/#Meat"/>
  <ov:new_Filler>
    <owl:Class>
      <owl:complementOf>
        <owl:Class>
          <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:resource="..#Meat"/>
            <owl:Class rdf:resource="..#Fish"/>
          </owl:unionOf>
        </owl:Class>
      </owl:complementOf>
    </owl:Class>
  </ov:new_Filler>
</ov:Modify_Restriction_Filler>
```

In this example, the value of the property “new_Filler” is a complex class definition. Note that “modification” operations, such as the one above, specify the new value as well as the old value. The old value is required to use the change specification for “undoing” changes, or when the operations need to be inverted.

A complete change specification often consists of several operations. We can group them together into one RDF document, using the Change_Specification class as a place-holder for the meta-data. A simple change consisting of two operations can look as follows.

```
<rdf:RDF>

<ov:Change_Specification about="">
  <ov:author>Michel Klein</ov:author>
  <ov:date>August 12, 2004</ov:date>
</ov:Change_Specification>

<ov:Set_Transtivity>
  <ov:from rdf:resource="..ontology/1/#larger"/>
  <ov:to rdf:resource="..ontology/2/#larger"/>
</ov:Set_Transtivity>

<ov:Add_Lowerbound>
  <ov:from rdf:resource="..ontology/1/#Car"/>
  <ov:to rdf:resource="..ontology/2/#Car"/>
  <ov:onProperty rdf:resource="..ontology/1/#hasWheels"/>
  <ov:new_Value>4</ov:new_Value>
</ov:Add_Lowerbound>

</rdf:RDF>
```

From the example it is clear that other elements of a change specification can be easily added. Composite operations can be added in a similar way, by grouping them into a "composite operation" class and connecting them via "consists_of" properties. In Chapter 8 we show how we build a change specification for a real ontology evolution. This study also contains examples of the representation of composite operations.

There are four assumptions with respect to the interpretation of the change specification.

- First, when the specification is executed, we assume that the "create" operations are performed before the other operations. This makes it possible to execute "modify" operations that use newly created concepts or properties.
- When concepts or properties that are mentioned in the "from" and "to" properties are only identified with a fragment identifier (not a complete URI, i.e. "`#Car`"), we look at the "source" and "target" values in the header of the change specification. The complete URI of the item in the "from" property can be found by attaching the fragment identifier to the value of the "source" property, whereas the URI for the item in the "to" property can be composed from the value of the "target" and the fragment identifier.
- We also assume that there exists an evolution relation between the concepts that are referred in the "from" and "to" properties. In the example above, there is an evolution relation between `Car` in version 1 and `Car` in version 2.
- Finally, if concepts or properties are not mentioned in the change specification, we assume an evolution relation between them when the fragment identifiers of both items in the different versions are syntactically equivalent.

5.6 Discussion

The language is just a specification mechanism and it does not impose restrictions on the actual change that is specified by its operations, neither on the correctness of the specification, nor on its efficiency. It is possible to use these constructs to specify incorrect transformations between two ontology versions, or transformations that are much more complicated than necessary. Also, inconsistencies in the specification can exists, e.g. the removal of a property restriction that didn't exist. The question how to cope with this is a procedural one and is not relevant for language itself.

One of the requirements for a change specification language was to allow for different granularity of the specification. Our language has this possibility because it can use both atomic operations and complex operations. Moreover, with the possibility to define ad-hoc composite relations, an arbitrary level of granularity can be achieved.

The change specification language that we propose is different from the versioning approaches that work with additions and deletions of RDF triples to and from the repository that is formed by an ontology definition (Ognyanov and Kiryakov, 2002). Our operations are at a higher level and contain more knowledge about the change. This results in more meaningful operations and more efficient specifications.

By using OWL for the ontology of change and RDF for the syntax of the change specification, we immediately have a large resource of tools that can be used process the specification. In Chapter 8, we show how we feed the ontology of changes and a actual change specification to a RDF query engine, which allows us to query for the effect of the change in a relatively easy way.

Chapter 6

Change Process

***Note**: Section 6.5 of this chapter has been published as part of an article in the proceedings of the IJCAI 2003 conference (Stuckenschmidt and Klein, 2003).*

In this chapter we describe how the evolution framework can be used to manage ontology changes. We start, in Section 6.1, with a description of the change process. In this description we explain how an ontology could evolve and how a change specification, as introduced in Chapter 4, can be *created* and *used* for performing ontology-related tasks.

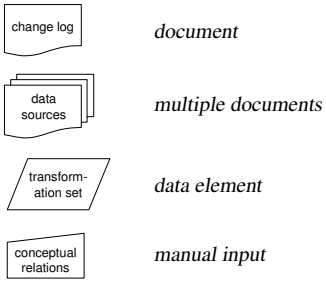
The creation process itself is detailed in Section 6.2. We provide a number of diagrams to explain how the different elements of a change specification can be derived and provided. For two of these processes, we present heuristics that can be used. After that we discuss for three different ontology-related tasks how the change framework can be used to solve some of the problems that are caused by evolving ontologies. The described tasks do not form a exhaustive list of possible ontology evolution tasks, but they address a number of the common problems that were identified in Chapter 3.

Section 6.3 describes the task of data retrieval and interpretation. It contains an analysis of different types of compatibilities between ontology versions and data sources, and a number of methods to (partly) interpret the data sources correctly. In Section 6.4, we describe how our framework can be aligned—and thus used—for the ontology synchronization methodology developed by Oliver (2000). Section 6.5 describes how the framework can be used for assessing the integrity of mappings between ontologies. Finally, Section 6.7 outlines some other possible usages of the change specification, e.g. editing support and collaborative working, visualization of changes, and for maintaining consistency.

6.1 Change Process Model

About the diagrams in this chapter

For the process diagrams in this chapter, we use a slightly extended flowchart notation. The rounded boxes represent activities, the diamonds represent choices and the solid arrows give the order of activities (the flow). The dashed arrows stand for input and output of data (knowledge, documents or manual input). In general, the data at the right side of the diagram are aids in the process, whereas the data at the left is the goal or result of the process. The uncommon icons are explained below.



In an uncontrolled and distributed setting, we assume that ontologies will evolve in a unpredictable way. Besides extensions and adaptations for specific purposes, ontologies will probably also be composed from parts of other ontologies. Figure 6.1 sketches an example of a possible ontology evolution process. An ontology A is extended with some definitions from ontology C, and the next two versions evolve in parallel. Each of the versions may continue to exist. In contrast with software products, where a new version often is meant to replace a previous version, there is not one "end product", but there will be many versions which might be all in use. As a consequence, the term *life cycle* is not appropriate for describing the development of an ontology; instead, we use the term *life trace*.

Not only the evolution process is unpredictable, also the use of ontologies can not be told beforehand. We cannot always assume a specific task for which the ontologies will be used, and we can also not guarantee that only the most recent versions of the ontologies will be used. As a result, an ontology-related task in a setting with evolving ontologies requires to determine the interplay of arbitrary versions for that specific task.

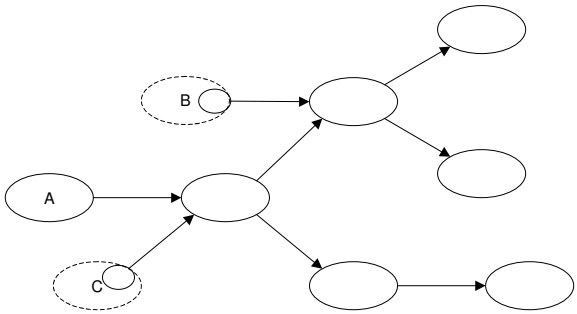


Figure 6.1: A possible evolution process of an ontology. The large ovals represent ontologies whereas the smaller ovals represent parts of ontologies.

Figure 6.2 specifies the change management process for evolving ontologies. Starting from a selected task, the first activity is to determine the specific versions from the ontology's *life trace* that are relevant for that task. The selection procedure is different for the respective tasks. For example, for change validation and approval—a task were a person

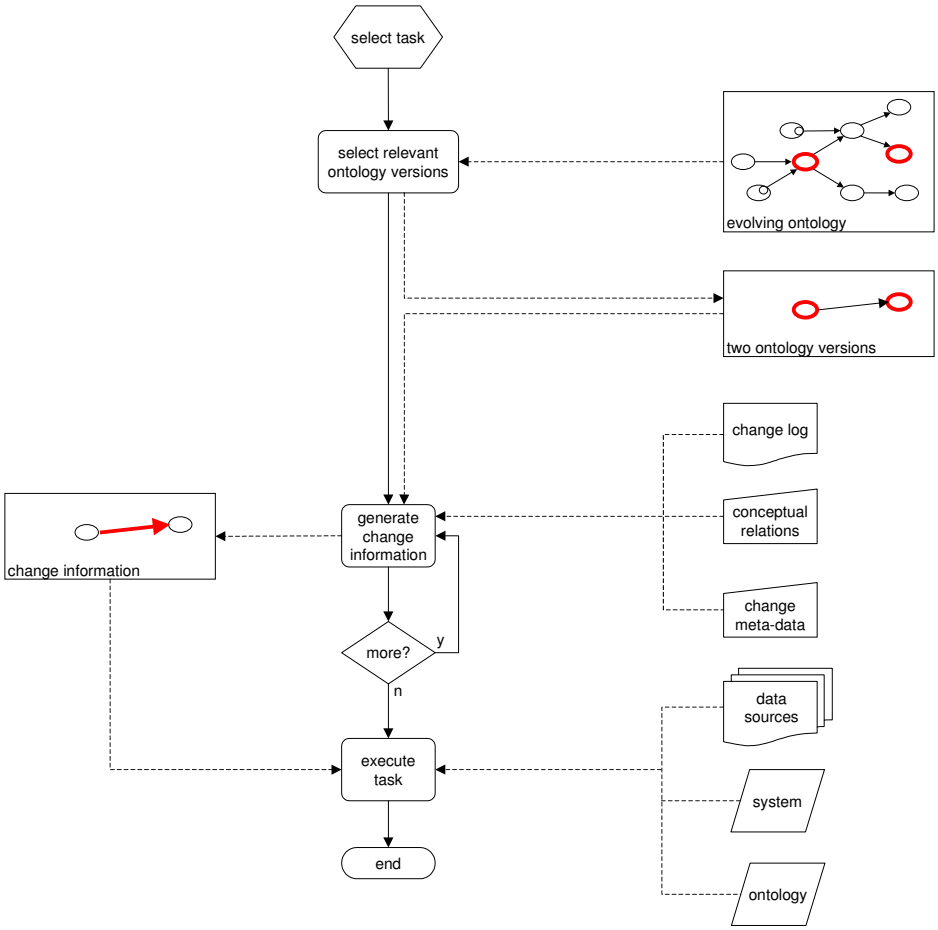


Figure 6.2: The process of managing ontology change for ontology-related tasks.

has to approve ontology changes that are proposed—it usually means that the newly created version and the original version are selected. For data retrieval, the version that is used to annotate the data and the version that will be used to retrieve the data should be selected. The result of this step is the identification of two ontology versions.

The next step consists of generating the change specification between these two versions. Dependent on the available information and the required knowledge for the tasks, new knowledge about the relation between the two identified ontology versions will be derived. The precise procedures for this are explained in the next section.

When enough information about the change between the identified versions is available, this knowledge can be used to perform the selected task, for example on data sources

or other ontologies. If, and to what extent the tasks can be performed successfully, depends on the knowledge about the change that is available in the end. In general, the aim of the change management methodology is to let tasks to be performed at least partially.

6.2 Creating the Change Specification

In Chapter 4, we have seen that there are a number of different ways to represent changes. As said, our assumption is that we do not know beforehand which information about the change is present. We either have the specification of both versions, or we have a change log with the new versions, or some conceptual relations, and so on. Based on editor logs or explicitly provided information, the change specification might also contain administrative meta-data and a rationale for the change. In this section we describe the processes to generate additional information about the change. Some of these processes are precise and others are based on heuristics.

To generate the additional change information, for each of the processes in this section should be checked whether the information required for its execution is available. This should be repeated until none of the processes can be executed anymore. Note that the output of some of the processes can be used as input for others; therefore, processes that could not be executed in the first round, could be executable in a next round.

6.2.1 Generating a Transformation Set

In many cases, it will be possible to create a transformation set between the two ontology versions. This can be done either by analyzing a log of an ontology editor or by comparing two ontology versions. Figure 6.3 specifies the steps and the choices. An editor-log could provide a bit more information than the two definitions of the ontology. Therefore, if both are available, using the log is preferable. For example, a log could reveal that a concept definition changed in several steps from X in V_1 to Y in V_2. The transformation set based on a log would than contain a set of change operations from X to Y. Based on the two ontology specifications, it might not be possible to discover that concept X turned into concept Y, so we would have an operation to delete X and another operation to add Y. Both transformation sets are correct, but the second contains less information. If a transformation set is produced from two ontology versions, we need to know which old definitions have evolved in which new definitions. Therefore, producing a mapping first is required.

Two tools for generating a transformation set from two ontology specifications are described in Chapter 7. Deriving a transformation set from a log basically consist of removing redundant operations and translating the editor operations into the vocabulary for the transformation set. We currently do not have an implementation of this procedure. In Chapter 8, we describe how we provide change information for a realistic series of evolved ontologies.

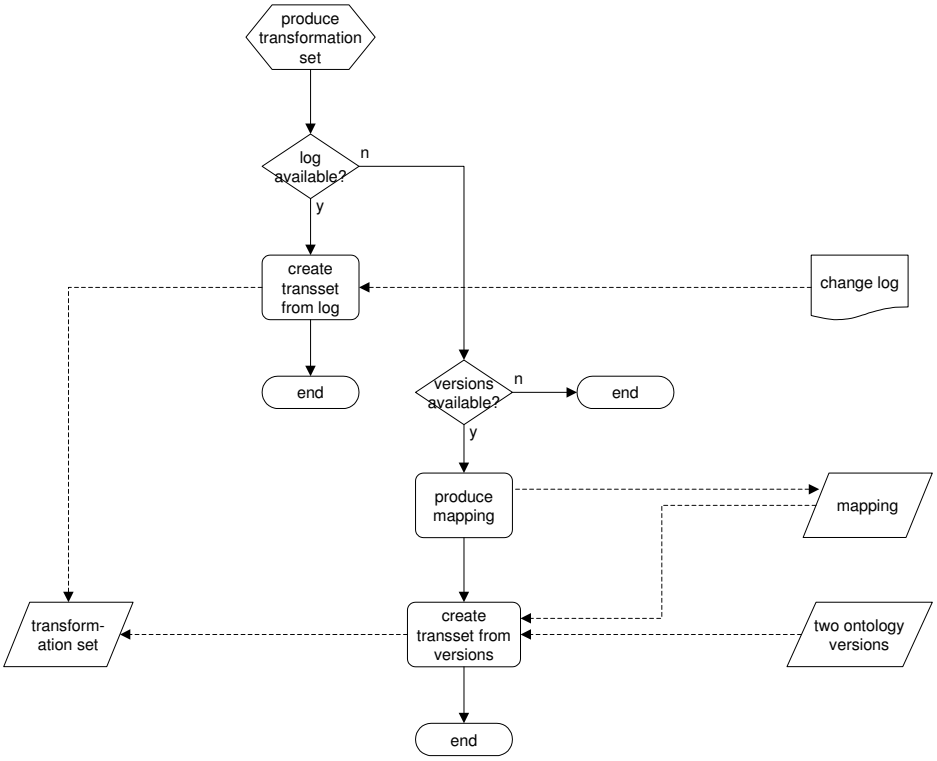


Figure 6.3: The process for generating a transformation set between two ontology versions.

6.2.2 Generating Two Versions

By its definition, if only one of the ontology versions is available, we can use the transformation set to generate the missing versions from it. For this, also the change log can be used. Figure 6.4 shows this process. The process can be executed in both directions: the old version can be re-generated from the new versions, and a new version can be generated from an old version.

6.2.3 Generating Complex Changes

When a transformation set has been produced, it can be processed to find complex operations, as is shown in Figure 6.5. We either use editor logs as a source for finding these operations, or the transformation set. Besides that, we might have to use both ontology versions to detect some specific complex operations.

Both the transformation set and the logs have their pros and cons as source for complex operations. A disadvantage of the editor-log is that it might lack the required ab-

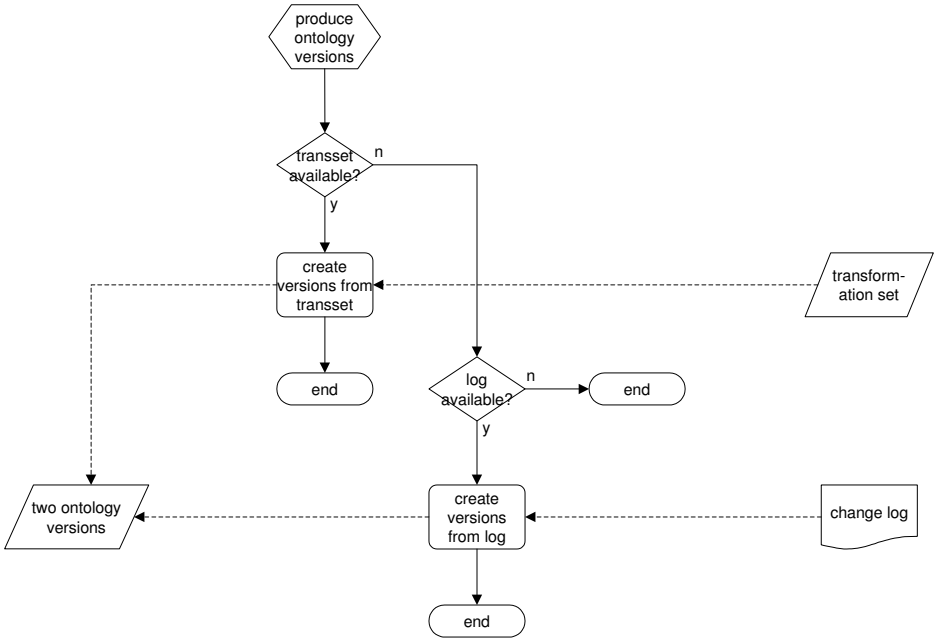


Figure 6.4: The process for generating two ontology versions from either a transformation set or a change log.

straction. For example, it could be that the editor log contains several non-successive moves of classes that result in a "subtree_move" in the end. This is not clearly visible in the log, but is easily detectable in the transformation set, where the order is not important anymore and the redundant operations are removed. On the other hand, for an advanced editor, the log file might already contain complex operations, such as "subtree_move". As with the previous process, we do not have an implementation for finding complex operations in a log. Providing a general procedure is difficult, as the procedure heavily depends on the (logging capabilities of the) editor.

To generate a complex change from a set of basic changes in the transformation set, we can sometimes use a set of rules. For other changes, we may not have a definitive set of rules for finding the complex changes and will need to use heuristics to determine if a complex change occurred. In the rest of this section we describe some rules and some of the heuristics that can be used. As is the case with the set of complex operations, the set of rules and heuristics is not complete and can be extended with additional ones.

Detection Rules

There are two types of rules to detect complex operations. The first type of rules uses the transformation set or the editor log file. Consider again the example of the wine

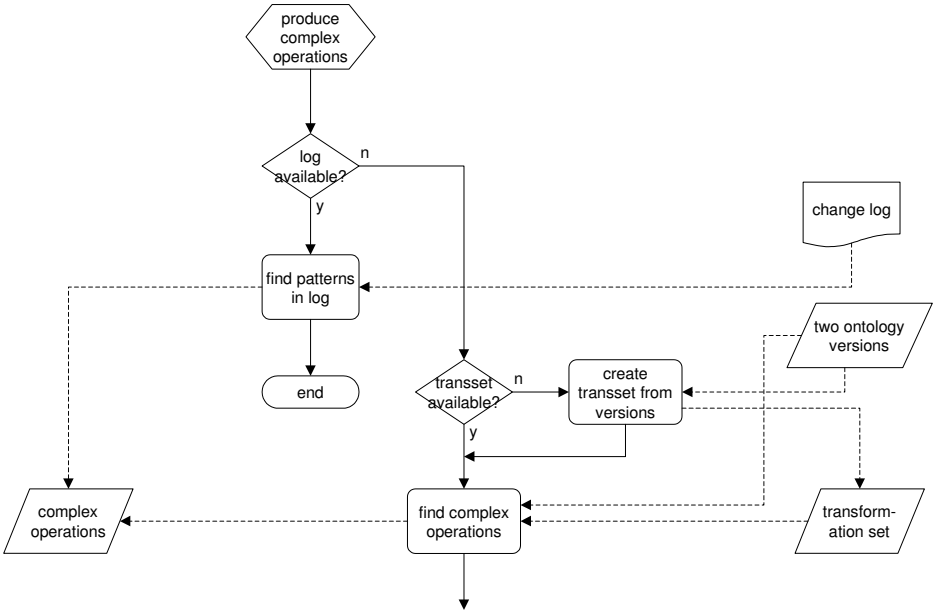


Figure 6.5: The process for generating complex operations.

ontology from Chapter 4 (especially Figure 4.3). In this example, three classes that were subclasses of White wine in V_{old} became subclasses of Rosé wine in V_{new}. We assume that we have an instantiated ontology of basic changes between the two versions. We can view this change as a set of several basic operations:

1. remove a superclass relation between Vin gris and White wine
2. add a superclass relation between Rosé wine and Vin gris
3. repeat the same for classes Cabernet blanc and White Ziinfandel

If we look at this set of operations conceptually, we can see that a complex operation was performed: a set of siblings was moved to a different location in the class hierarchy. There are two levels of "enrichment" that we can identify in this example. First, we can recognize the "add superclass"–"remove superclass" sequence for each of the classes Vin gris, Cabernet blanc, and White Ziinfandel as a move in the tree. Second, we can recognize that Vin gris, Cabernet blanc, and White Ziinfandel were and remain siblings in the class hierarchy and thus we have a "move siblings" operation.

The set of rules to recognize this change is rather simple: as long as we know that the class A in the V_{new} is the same as the class A in V_{old} (this information is readily available in a mapping) and their superclasses are different, we can identify a "move" operation. To recognize that a set of siblings was moved together, we compare arguments of the

"move" operations. If the to and from arguments for a set of "move" operations are the same, we have a "move siblings" operation. We can summarize the rules for determining a move of siblings to a new place in a hierarchy with the following three rules:

1. A class $C \in V_{old}$ has n direct subclasses: $subC_1, subC_2, \cdots, subC_n$
2. There exists a class $newC \in V_{new}$ and more than one of its subclasses belong to the list $subC_1, subC_2, \cdots, subC_n$;
3. These new subclasses of $newC$ are no longer subclasses of C (i.e., no multiple inheritance).

A second class of rules does not just use the transformation set to detect complex changes, but also accesses the ontology versions V_{new} and / or V_{old}. Suppose we know that a range of a property P was changed from C_1 to C_2. If we have access to V_{old}, we can check whether C_2 is a subclass of C_1 in V_{old}. If it is, then the range of the property P was restricted. As a result, for example, some instances that use this property may become invalid. On the other hand, if we know that C_2 is a *superclass* of C_1 in V_{old}, we also know that the range of the property P has become less restrictive (another complex operation). If our task is data interpretation, we can conclude that no instances were affected. Therefore, if all we have, for example, are V_{new} and a transformation set with basic operations, these are the complex operations we will not be able to identify. Restricting a range of a property is an example of such an operation.

In Chapter 7, we describe a tool that uses the mapping between concepts and the two ontology versions V_{old} and V_{new} to identify a number of those complex changes.

Detection Heuristics

In the ideal case, we can use rules to precisely identify complex changes that involve multiple classes, e.g., a group of sibling classes that was moved to a new place in the class hierarchy. However, if other changes have been made between the same versions, the conditions for the rule might not be met, for example because one of the classes involved has been deleted later on. In other words, while in principle we can specify a precise set of rules to determine when a complex operation occurs, in practice additional changes in the ontology involving the same concepts, may make a decision that a complex change has occurred less clear-cut.

Consider for example the following operation: group a set of siblings to create a new superclass (create a new abstraction). We would have such an example if we grouped the Rosé wines in Figure 4.3a together to create the Rosé wine class, but have left this new class as a subclass of White wine.

In the ideal case, we have the following conditions that describe the case when such an operation has occurred:

1. A class $C \in V_{old}$ has n direct subclasses: $subC_1, subC_2, \cdots, subC_n$
2. There is a class $newC \in V_{new}$ such that:
    - $newC$ is a direct subclass of C

- $\forall\ subC \in V_{new}$ such that $subC_i$ is a direct subclass of $newC$, $subC_i$ was a direct subclass of C in V_{old}

However, the user may have also, for example, added other subclasses to $newC$ (e.g., adding new types of rosé wines). He may have also added another level of classes between C and $newC$.

To cope with these kind of problems, we have to replace the exact rules by heuristics that allow for some noise in the set of change operations. This can be done by changing the precise criteria to approximate criteria. For example, we can rephrase the conditions for $newC$ above resulting in a heuristic of the following form:

- $newC$ is a subclass of C (not necessarily direct)
- Among the direct subclasses of $newC$, *most* come from $subC_1, subC_2, \cdots, subC_n$

To make this practically usable, the values for the approximate criteria such as *most* need to be determined empirically. This could lead to computable criteria such as "more than 50% of subclasses of $newC$ must be former subclasses of C" or "there is at most one level of classes between C and $newC$".

6.2.4 Generating Evolution Relations

Figure 6.6 shows the process for finding the *evolution relation*, i.e. the mapping between corresponding constructs in the old version and the constructs in the new version of the ontology, specifying which concept has evolved in what other concept. Basically, there are three options for generating the mappings. If the ontology contains persistent identifiers for concepts and relations, those can be used. If not, then the change log can be used to find the mappings. Changes in the identifier of a concept are a common cause that prevents mappings from being found. A change log, however, will often explicitly contain the information that an identifier of a concept has changed, and thus provide the "trace" of the changes between two versions of a concept. If a log is not available either, then mapping heuristics have to be applied. Such heuristics are similar to those that are used for schema matching (Rahm and Bernstein, 2001), but they are normally much more tolerant, in the sense that less evidence is required for a match to be accepted. The requirements can be lowered because the chance for two concepts that are similar to be a match is much larger when we assume that the ontologies are derived from each other, than for unrelated schemas. PROMPTdiff (Noy and Musen, 2002) is a tool that implements around 10 heuristics for creating mappings between concepts in different versions of ontologies.

6.2.5 Generating Conceptual Relations

The conceptual relation between two concepts specifies the intended semantic relation between the two versions of a concept or relation, i.e., whether the changed version should be considered equivalent to the original version, or that it subsumes or is subsumed by the other. As is explained in Section 4.1.1, this is in the end an interpretation of the person that will use the changed concepts.

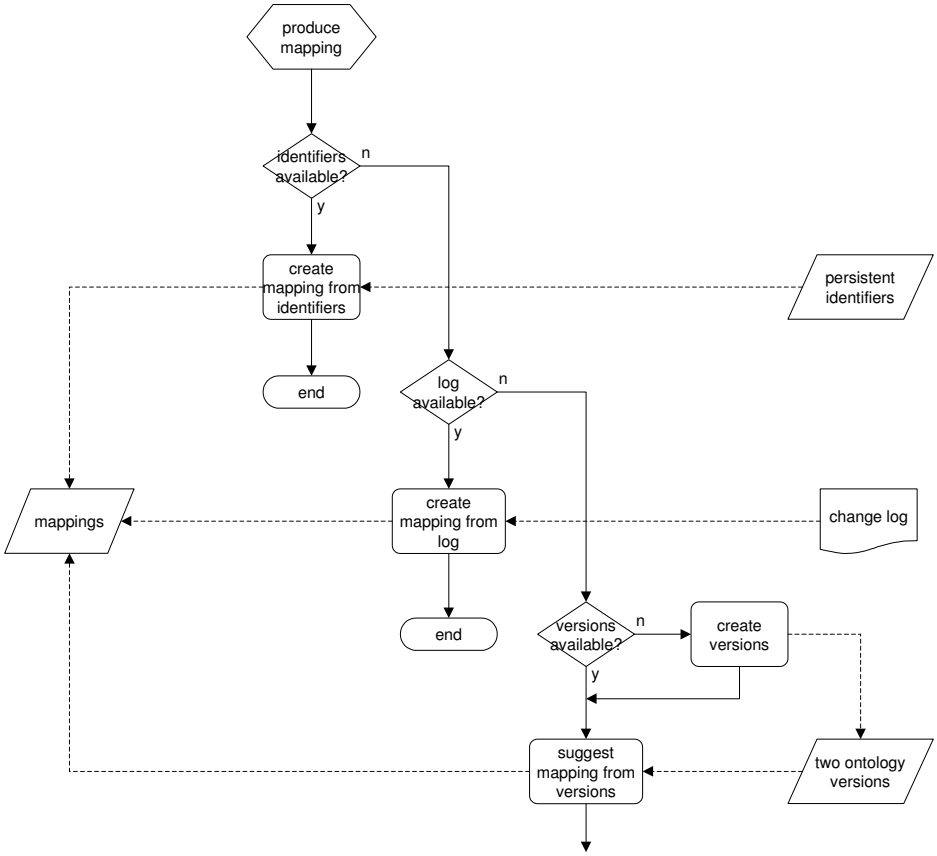


Figure 6.6: The process for generating a mapping between constructs in different versions of the ontology.

The process for finding conceptual relations is specified in Figure 6.7. There are many possible inputs for generating conceptual relations. First of all, if both versions of the ontology are available and the semantics of the ontology can be expressed in a Description Logic (Baader et al., 2003)—as is e.g. the case with OWL DL—the exact subsumption and equivalence relations according to the definitions can be computed automatically. Examples of reasoning systems that are capable of doing this are FaCT (Bechhofer et al., 1999) and RACER (Haarslev and Moller, 2001). However, the computed subsumption relation between concepts might not always be the intended interpretation of the conceptual relation. Consider the example from Section 4.1.1, where an additional property restriction “fuel-type” is added to the class “Car”. If modeled as a defined class, a reasoner will derive that the new version of car is a subset of the old version (see Figure 6.8), although the person who performed the change might consider them as

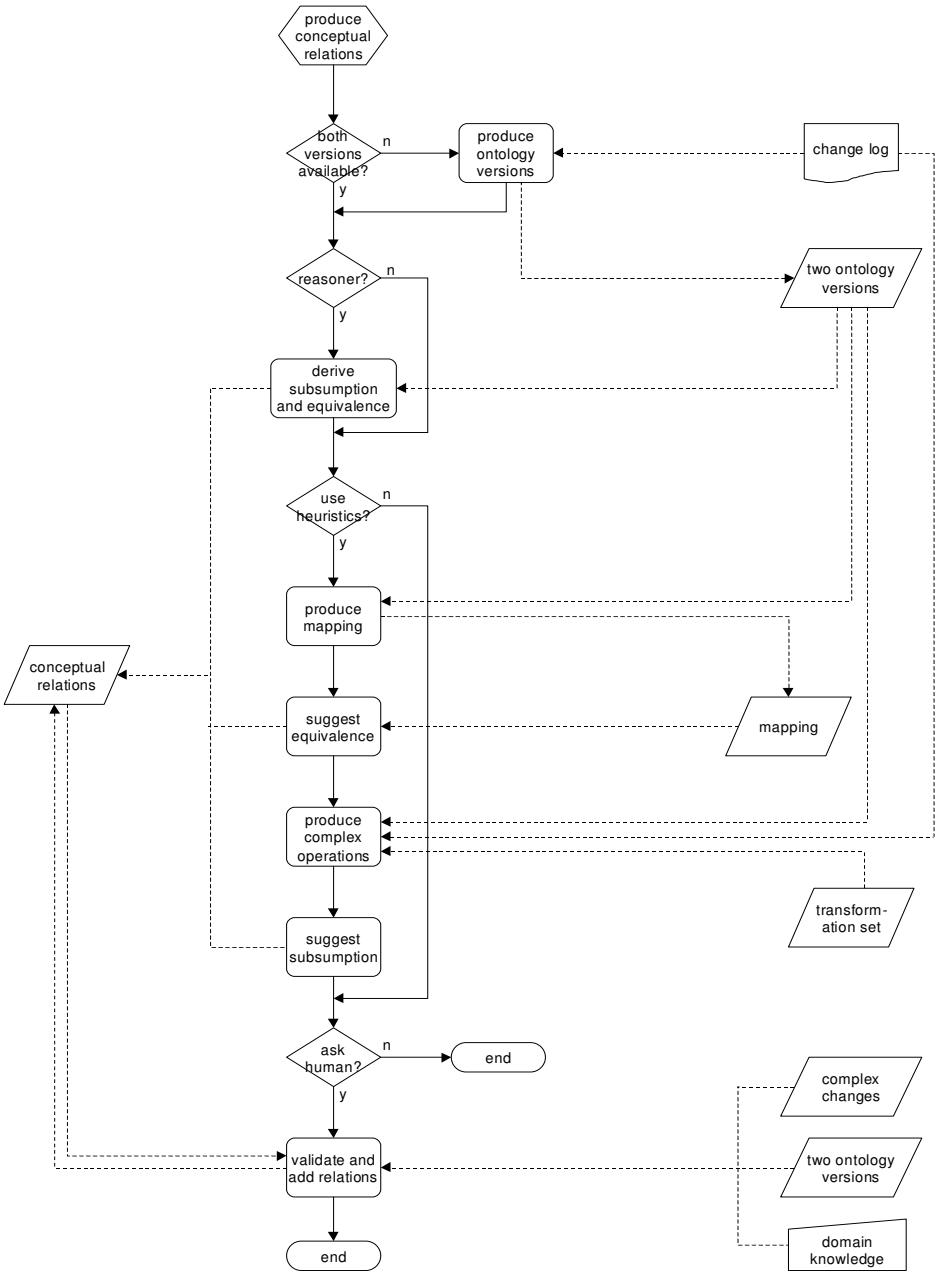


Figure 6.7: The process for generating conceptual relations between constructs in different versions of the ontology.

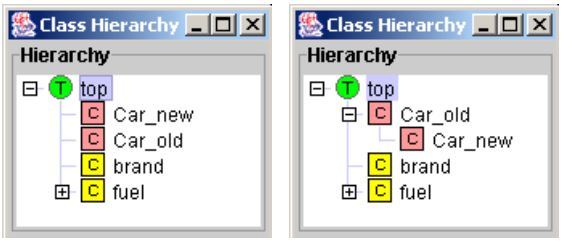


Figure 6.8: A tiny subsumption hierarchy with two versions of the definition of "Car" before and after classification.

equivalent. Therefore, validation of the derived relation by a domain expert is necessary.

Heuristics

If an exact computation of the logical relation can not be done, heuristics can be used to suggest conceptual relations to a human expert that understands the domain of discourse.

There are two sources for such suggestions. First, based on the *mapping* between concepts in the old version of the ontology and those in the new version, suggestions for equivalence relations can be made. If it is known that a specific concept in one version of the ontology maps onto a concept in another version, there is basis to assume that there exists an equivalence relation between both concepts, especially if there is no difference in the definition of the concept.[1]

Second, if there is a difference in the definition, we can use the *transformation set* to suggest conceptual relations between both versions of the concept. Based on our observations about the possible effect of changes to concepts in their place in the hierarchy, we present in Table 6.1 a number of suggestions for the conceptual relation between the old and new concept for specific change operations.

operation	x **in** C_{new} x C_{old}
Add_Property_Restriction	$\subseteq$
Remove_Property_Restriction	$\supseteq$
Add_Label	$-$
Add_Equivalent_Class	$\supseteq$
Remove_Equivalent_Class	$\supseteq$
Add_Superclass	$\subseteq$
Remove_Superclass	$\supseteq$
Extend_Cardinality	$\supseteq$
Restrict_Cardinality	$\subseteq$

Table 6.1: Heuristics rules that suggest the the effect of respective operations on conceptual relation between the new concept C_{new} and the old concept C_{old}. A "$-$" means that the operation does not provide basis for a suggestion about the resulting conceptual relation.

[1]Note that we use the assumption that two concepts with the same name but in a different name space are different until they are explicitly said to be equivalent.

When a change is described with multiple operations, a suggestion for a conceptual relation can only be done if all operations result in the same suggestion; if not, the expected effect of the respective operations is probably opposite and a useful suggestion is not possible.

6.3 Retrieval and Interpretation of Data

We now turn our attention to the usage of the change specification for specific tasks. In this and the next three sections, we describe the process for a number of those tasks.

A very prominent application of the ontology change framework is data retrieval and interpretation. In such a scenario, we assume that ontologies are used to describe the "meaning" of pieces of data. Computers use the knowledge in ontologies to combine and relate the data in a—from a human stance—meaningful way. This is typically the situation that is envisaged for the Semantic Web (see the introductory chapter).

In such a situation, one can think of at least two different tasks that can be performed:

data retrieval: for a (combination of) term(s) in an ontology, retrieve the pieces of data that are instances of it;

data interpretation: for a given piece of data, get the most specific concept or relation that is accurately describing it.

In this section, we describe how the ontology evolution framework can be used for data interpretation and data retrieval. However, before we can do this, we first have to analyze the precise problem that occurs when performing these tasks. This is done in Section 6.3.1, where we discuss the issue of compatibility. After we have done this, we specify two mechanisms to cope with changing ontologies when performing these tasks. The first one is a method to determine whether a changed ontology can be used instead of the original one (Section 6.3.2), the second mechanism is an approach to partly translate data sources.

6.3.1 Compatibility of Changed Ontologies

Changes in ontologies can cause that data cannot be interpreted or retrieved correctly anymore, because the version of the ontology might not match the version of the data source anymore. In general terms, this is called incompatibility.

Types of Compatibility

We first define precisely what we mean with "compatibility". In general terms, compatibility is *the capability that allows the substitution of one subsystem (or functional unit), for the originally designated system (or functional unit) in a relatively transparent manner, without loss of information and without the introduction of errors*.[2] This definition

[2] Slightly adapted definition from the Wikipedia encyclopedia, based on the Glossary of Telecommunication Terms, Federal Standard 1037C.

mentions two different requirements for a compatible substitution: no information loss, and correctness. In the setting that we consider, we can make this definition more precise. For data interpretation, in addition to the correctness, the *precision* of the interpretation is an aspect to consider. If pieces of data are to be interpreted, it could be that they are correctly typed, but not with the most specific terms possible. For data retrieval, we can distinguish between two different variants of the 'no information loss' requirement. The first interpretation is that a query, using concepts and relations from the ontology, gives exactly the same answer for two versions of an ontology. The second interpretation is that all information is accessible via both versions of the ontology, but not necessarily via the same query.

Altogether, this results in four different interpretations of compatibility of ontologies and data:

A: being able to use another version of the ontology to retrieve all data using the same query;

B: being able to use another version of the ontology to retrieve all data;

C: being able to use another version of the ontology to interpret all data correctly and as precise as possible;

D: being able to use another version of the ontology to interpret all data correctly.

Besides these variants of compatibility, it is also of interest to know whether changes that produce incompatible versions of ontologies, actually introduce errors. If not, the incompatible ontologies are still usable to retrieve or interpret *part* of the data correctly. We call this kind of relation between ontology versions and data sources *incomplete compatible*.

Directions of Compatibility

Compatibility between ontologies and data sources can be considered in two different directions: in *prospective use* and in *retrospective use*. These terms, which are borrowed from database schema versioning literature (Roddick, 1995), are illustrated in Figure 6.9. "Prospective use" is using a newer version of an ontology with a data source that conforms to an older version, and "retrospective use" is using an older version of an ontology with a data source that conforms to a more recent ontology. Unlike with some more controlled environments, in an uncontrolled environment we cannot expect that one incompatibility is more important than the other. More specifically, in database systems the most important direction is often prospective use, i.e. using a new database schema to access older data. In a distributed setting, however, both directions of incompatibility should be taken care of.

When the change to an ontology results in a revision that can be used prospectively, it is called a *backward compatible* change. This is the case when the modification in the ontology does not affect the existing definitions, i.e., when the change is a monotonic extension. Heflin and Hendler (2000) show that the addition of concepts or relations are such extensions. When used on a data source, ontologies that are extended in this

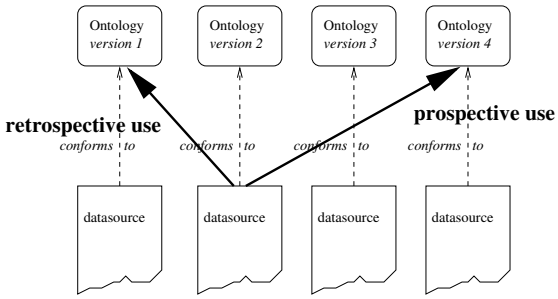


Figure 6.9: Examples of prospective and retrospective use of ontologies.

way yield the same perspective as when the original ontology is used. Changes are called *upward compatible* when data sources that obey the new ontology can be used retrospectively. For example, this is true for a deletion of an independent class.

Notice that both backward compatibility and upward compatibility are transitive: when the changes from $v1$ to $v2$ as well as the changes from $v2$ to $v3$ are backward compatible, then the changes from $v1$ to $v3$ are also backward compatible.

Consequently, if we know that all subsequent revisions to an ontology up to a certain version are backward compatible, it is also possible to name the resulting version of the *ontology itself* backward compatible. However, it is never allowed to call a version of an ontology *upward compatible*, because the semantics of future versions are not known beforehand. It is always possible that new versions of ontologies will introduce new things that cannot correctly be interpreted via older ontology versions. Thus, *backward compatibility* can be a characteristic of an ontology, but *upward compatibility* can not be.

The change management framework can help to determine the type of compatibility between versions of ontologies and to translate data to achieve a maximal data accessibility.

6.3.2 Determining Compatibility

To determine the compatibility between different versions of ontologies, we look at the effect of the different operations in the transformation set between two ontology versions. The complex operations allow to describe that combinations of basic operations can have a less destructive effect on the compatibility when applied together than the sequence of basic operations themselves.

Earlier, we have given the example of a slot that is “moved up” in the hierarchy. If we treat this operation as a sequence of two operations, removing the slot from the subclass and then adding it to the superclass, the instance data of the slot is lost in the first operations. However, the composite effect of the two operations does not violate the integrity of the instance data.

To give an impression of the effects, Table 6.2 lists a number of the operations that are defined in Chapter 5 and describes the effects on the different types of compatibil-

ity of ontologies. The characters in the second column of the table refer to variants of compatibility in the list above. If the character for a specific compatibility is present, it means that the specific type of compatibility is broken by that operation. Note that the table specifies "worst-case scenarios". It tells that a specific compatibility cannot be guaranteed anymore, but this does not mean that *every* combination of data and the two ontology versions are incompatible. For example, in general deleting a concept is incompatible for data retrieval and precise interpretation (variants A, B and C), but if a specific data source does not contain instance data for that concept, it does not harm the interpretation. However, in an uncontrolled and distributed setting we cannot assume that we know something about the data itself.

Note that in this analysis we assume that there are only explication changes, i.e. that the conceptualization itself is not changed. In case there are conceptual changes as well, we cannot guarantee the compatibility and have to use the explicit conceptual relations between the version of constructs to re-interpret data (see next section).

Operation	**Broken**	**Explanation**
Add_Class	C	The added class might describe the data more precisely.
Remove_Class	ACD	Instances of the class have a less specific type, but can always be retrieved via the most general class.
Add_Property	-	All data can still be retrieved and interpreted.
Remove_Property	ABCD	If there exists a value for the removed property, we don't know anything about the value anymore.
Add_Property_Restriction	ABCD	Old values for properties might not be valid anymore.
Remove_Property_Restriction	-	Everything which was valid before is still valid.
Modify_Superclass_To_Superclass ("move a class up in the hierarchy")	-	There can be fewer inherited property restrictions for the class, but all properties are still known.
Modify_Superclass_To_Subclass ("move a class down in the hierarchy")	AB	There can be additional inherited property restrictions which could invalidate some data.
"Widen" a property (restriction), e.g., Increase_Cardinality or Change_Range_to_Superclass	-	All values for the property are still valid.
Delete_Class_And_Move_Siblings_Up	ACD	Instances of the deleted class itself have a less specific type

Table 6.2: The effect of change operations on different types of data compatibility.

The table lists the effect on retrospective use, i.e. it specifies whether the *backward* compatibility can be maintained or not. To determine the effect on prospective use, there are two options. First, the transformation set between the two version of the ontology can be calculated in the other direction. This means that, instead of deriving the required operations between V_{old} and V_{new}, the operations to transform V_{new} into V_{old} should be calculated. The other option is to determine the *inverse* for each operation, and look up the effect for these operations. The inverse of "add" operations are "delete" operations

(and vice-versa), and the inverse of "modify" operations are "modify" operations with the "from" and "to" argument exchanged. The change_to_superclass and change_to_subclass are also each others inverses.

6.3.3 Partly Translating Data

In the section above we described how the change operations can be used to determine the compatibility when a changed ontology is used instead of the original ontology in a specific direction and for a specific task. If the ontology is compatible for a specific task, this means that the changed ontology can be used instead of the original ontology without loss of information and without introducing errors. If not compatible anymore, we might still be able use the ontology to retrieve or interpret data. Below we will describe a procedure to retrieve data via ontologies that changed in a way that we called *incomplete compatible* above.

For data retrieval, we not only exploit the change operations, but also the conceptual relations between the concepts in the old and the new version of the ontology. We assume that we have simple conjunctive queries without negation. It requires further investigation to know how this approach can be applied in the case of more complex queries. The method is as follows:

1. for each changed concept or property, look up whether a conceptual relation between the original and the new version is specified.
    - if an equivalence relation exists, replace the concept in the query with the equivalent concept from the old ontology;
    - if a subsumption relation exists:
        - if the new concept is superclass of the old concept, replace the concept in the query with the concept from the old ontology;
        - if the new concept is subclass of the old concept, remove the concept from the query.
2. if the concept has been deleted, replace the concept in the query with the union of the subclasses of the concept from the old ontology.
3. if the property has been deleted, replace the property in the query with the union of the subproperties of the property in the old ontology.
    - if no subproperties existed, remove the related concepts from the query;
4. if the operation involves a merge of the concepts, use union of old concepts to replace new concepts;
5. if the operation involves a split of the concept, remove the concept form the query.

For interpreting data, a similar algorithm can be used, which is basically an inverted version of the algorithm above.

6.4 Ontology Synchronization

The term *ontology synchronization* is introduced by Oliver (2000). She defines synchronization as "the periodic process by which developers update the local vocabulary to obtain the benefits of shared-vocabulary updates, while maintaining local changes that serve local needs". This process is very common in the health-care domain, where local hospitals use adapted versions of national or international terminology standards. Keeping the local versions up-to-date with the evolving global version is necessary to stay current with new insights and for being able to exchange information with other users of the vocabulary.

In this section, we describe the approach for ontology synchronization developed by Oliver, called CONCORDIA, and relate the underlying models in her methods to elements in our framework. We then show how her synchronization methods can be applied within our framework for ontology evolution.

6.4.1 CONCORDIA Synchronization Approach

The CONCORDIA synchronization approach is developed specifically for medical vocabularies. Consequently, it makes a number of assumptions about the structure of the ontology and the development process. First of all, the CONCORDIA methods assume that there is one *shared* version of an ontology and one or more *local* versions of an ontology that are derived from the shared one. Second, the one and only goal of the approach is to bring the local versions in line with the shared version. This basically means that the local versions should be made pure *extensions* of the shared version. Third, it assumes that every concept has a constant and unique identifier, which is separate from a meaningful name that can change and can possibly have synonyms.

The knowledge model of CONCORDIA consists of three elements. The first element is called the *structural model*. In essence, this model defines the meta model for the vocabularies. It defines how concepts are defined, the relations between them, and the elements of attributes. In Section 6.4.2, we show the precise models as we compare them with the elements in our framework. The second element of the CONCORDIA model is called the *change model*. This model defines all different changes that can be applied to vocabularies, distinguishing between changes to local vocabularies and changes to the shared vocabulary. Thirdly, the *log model* defines an exchange format for changes.

The CONCORDIA change process is as follows. Over time, change operations that are defined in the change model can be applied by the developers of the vocabulary versions, which results in a divergence of the shared and local versions. Once in a while, the vocabularies will be synchronized. Oliver has defined a *synchronized state*, which specifies the desired target state of the vocabularies. In this state, both every concept, and every subsumption relation between concepts, and every attribute–value pair in the shared vocabulary should also be present in a local vocabulary. In addition to this, the concepts should have the same unique identifiers in both versions. The synchronization is performed by processing the log of the changes that are made in the shared version during the time period between the last synchronization actions and the current moment. For each of the changes in the log, a list of *action choices* is defined. Each of those actions

defines a sequence of steps that should be performed to bring the local vocabulary in a synchronized state w.r.t. a specific change in the shared vocabulary. The choice between different actions is partly based on the conflicts with the structural model that are possibly introduced by the actions. For example, for adding a concept, a criterion is whether or not a concept with the same name already exists. If this is the case, an action should be chosen which also renames one of the concepts.

6.4.2 Alignment with Change Framework

The CONCORDIA methodology uses a different knowledge model than the OWL-based knowledge model that we assume in our framework (see Section 5.3). Also, the assumptions about the evolution process are restricted to a specific scenario. However, for a subset of all possible evolution scenarios and ontologies, the CONCORDIA synchronization methods are applicable within our framework as well. Although the terms are different, the basic elements that are present in CONCORDIA also exist in our framework. In the next paragraphs we will compare the elements of both approaches. We then show how the CONCORDIA approach can be performed within our framework and discuss the extension that should be made to make it applicable to a broader set of ontologies.

Ontology Meta Model

What is called "structural model" by Oliver, is the meta model of the ontology language for us. The meta model that is used in CONCORDIA is designed specifically for medical vocabulary and therefore less general than what we use. To apply the synchronization methods in our framework, we will consider the *minimally required* elements and express them in our knowledge model. The CONCORDIA approach requires that a shared *concept* consists of at least: 1) a unique identifier, 2) a name, 3) a non-empty set of parents, and 4) a usage status. We discuss how each of these concepts can be represented in our framework.

In OWL, URI's (Berners-Lee et al., 1998) are used as unique identifiers. Each concept and property (except for anonymous concepts, which are only used as building blocks for other named entities) is identified with a name, syntactically often represented as a fragment identifier, e.g., "`#Book`". In the RDF/XML framework, such fragment identifiers are transformed into a URI reference by appending the fragment identifier to the in-scope base URI (as described in Beckett, 2003). Thus, each concept and property has a unique identifier in the form of a URI.

The name of concepts and properties often coincide with the fragment identifier that forms part of the unique identifier. This conflicts with the requirement in CONCORDIA that the unique identifier is persistent and unconnected with the meaningful name. However, by using the `rfds:label` attribute for concepts and properties, we can give them a name that is not connected to the identifier. To be compatible with the CONCORDIA model, we also have to require that the identifier will not change.

The parents of a concept in OWL are specified via `rdfs:subClassOf` statements. Although these statements are optional for OWL class definitions, the requirement that

every concept has at least one parent is met by the fact that every OWL class is a subclass of `owl:Thing`.

The usage status is something that is not present in OWL. However, via the mechanism of "annotation properties", we can define additional attributes of concepts and properties. With the following definition, we can add the usage status to the OWL meta model.

```
<owl:AnnotationProperty rdf:about="#usageStatus">
  <rdf:range>
    <owl:Class>
      <owl:oneOf rdf:parseType="Collection">
        <owl:Thing rdf:about="#current"/>
        <owl:Thing rdf:about="#retired"/>
      </owl:oneOf>
    </owl:Class>
  </rdf:range>
</owl:AnnotationProperty>
```

A specific class definition would use this property as follows:

```
<owl:Class rdf:about="#Book">
  <rdfs:label>Book</rdf:label>
  <cc:usageStatus rdf:resource="current"/>
</owl:Class>
```

In addition to concepts, the CONCORDIA model also contains properties, called *attributes*. Attributes consist of at least: 1) a unique identifier, 2) a name, and 3) a usage status. All these elements can be expressed in our model as well, in a way similar to the way in which their counterparts in concepts are expressed.

The knowledge model of a *local* vocabulary is an extension of the model of the shared vocabulary. Both concepts and attributes have an additional property "site-of-origin", whose value is either "shared", "locally modified" or "local-only". Moreover, the value of "usage-status" can also be "hidden" or "preserved", and there is an additional list of superclasses that specify the superclasses of a concept in the *shared* vocabulary.

Change Operations

The "change model" in CONCORDIA is the equivalent of our ontology of change operations. There is one major difference between both models. In CONCORDIA, "delete" operations do not exist because all concepts are persistent. Instead, there are two different variants of removal for both concepts and attributes. In a shared vocabulary, a concept or attribute can be "retired", which means that it is not meant to be used for future annotations anymore. In a local vocabulary, concepts and properties will be "hidden". This means that they are still available in the shared vocabulary, but that they are not used at the local site.

Besides this, the change operations also differ on some minor points. In CONCORDIA, there are a few change operations that are specific for its structural model, such as "add translation code" and "delete synonym". Because these operations refer to optional elements of the structural model, we can ignore them for our purpose of expressing the minimally required elements. Also, there are two types of "merge" operations: in one

variant, the resulting concept is the same as one of the two old concepts, in the second variant, the two original concepts are merged into a new one.

We incorporate the additional CONCORDIA change operations in our framework by extending the ontology of change operations with some new complex operations. Note that a basic assumption in our framework is that the set of complex changes can always be extended if this is desired for a specific purpose. Table 6.3 shows how CONCORDIA changes that are not in our ontology of operations can be expressed as complex operations.

CONCORDIA **operation**	**Complex operation**
Retire concept	retire_class subclass of class_change consists of: • add_annotation with property "usageStatus" and value "retired" • modify_subclass for each subclass to the parent
Retire attribute	retire_property subclass of property_change consists of: • add_annotation_property with property "usageStatus" and value "retired"
Merge two concepts into one of the two concepts	incorporate_class subclass of merge_class consists of: • merge_class for A and B into C • delete_class for B • add_class_equivalence for A and C
Correct concept name	replace_name subclass of class_change consists of: • modify_annotation_property with property "label" and value "new name"
Replace concept name	replace_name subclass of class_change consists of: • modify_annotation_property with property "label" and value "new name" • add_annotation_property with property "synonym" and value "old name"

Table 6.3: Operations from the CONCORDIA change model that do not have a direct counterpart in the ontology of change operations, and their specification as complex operation.

Change Representation

The "log model" in CONCORDIA is the counterpart of our change representation *language*. In CONCORDIA, the log is used to represent the changes that have occurred in the shared vocabulary. The log is an ordered sequence of change records, where each

of the records contains one change operation, its operands, and meta-data such as a time stamp, natural language explanation, and author information. For each of the elements in the change record, a one-to-one mapping to elements in our change representation language can be made. The only difference is the maintenance of order. In a change log in CONCORDIA, the order is explicitly represented, while a change representation in our framework contains a set of operations, where we require that "create operations" are executed before other operations. For the purpose of re-executing changes in an other ontology where the identifiers are persistent, the latter requirement gives the same result as keeping the complete order. In short, the change representation in our framework can be used as basis for ontology synchronization in Oliver's methodology if the "create operations" are parsed before the other operations.

6.4.3 Discussion

In the above paragraphs we have described how the minimally required elements of the CONCORDIA synchronization methodology can be expressed in our framework. Doing this, we have shown that the synchronization approach can be applied for vocabularies that follow the assumptions of Oliver when they are expressed in OWL and their evolution is described within our framework. This allows us to perform ontology synchronization for a specific subtype of ontologies.

A question that follows from this is whether the described approach can be extended towards a generally applicable synchronization methodology. This means the approach should neither require any extensions of the knowledge model of the ontology, nor should it make any assumptions about the usage. To answer this question, we have to analyze what the precise goals of the methodology are, and which of its elements are necessary and which not.

The CONCORDIA synchronization procedure appears to have two related goals. One is to update the local vocabulary with changes that have occurred in the global vocabulary, the other is to maintain a number of invariants of a synchronized ontology. These invariants are described in the definition of a *synchronized state*. Combined with the fact that the knowledge model in CONCORDIA should meet the requirements for expressing medical vocabularies, this gives three different origins for elements behind the methodology. They either enable:

1. expressing medical vocabularies, or;

2. maintaining particular invariants, or;

3. re-executing the global changes.

For a generic ontology synchronization procedure, the elements that have their origin in the first two goals are not relevant. This means that the extension to the knowledge model for representing meaningful names is not strictly required. The same holds for some of the steps in the "action choices" that have the specific goal to guarantee that each concept and subclass relation in the shared ontology is available in the local ontology.

The elements that are required are those that allow the re-execution (within the local ontology) of changes that are performed in the global ontology. The basic principle behind these elements is that it should always be possible to trace back the original concept in the local vocabulary. This is implemented via three main mechanisms: 1) having persistent identifiers, 2) not being able to delete something (but instead flagging concepts as "hidden" or "retired"), and 3) update procedures in the "action choices".

Unfortunately, the first two mechanism can not be implemented in an unrestricted setting. It cannot be guaranteed that identifiers are persistent. Also, it is not possible to prohibit deletions, and for flagging the status of concepts an extension of the knowledge model is required. Therefore, in general the synchronization approach is not applicable in an unrestricted setting.

However, one could think of other ways to achieve the same principle, i.e. being able to trace back the original concept. The problem of having no persistent unique names can be partly solved by exploiting the evolution relation. The goal of a persistent identifier is to be able to trace the evolution of a concept. If a first step is added to the procedure which finds the predecessor of the concept, the concept can be traced back. One of the problems with deletions is that local changes on globally deleted concepts might get lost as well. This could possibly be addressed by a procedure that first reverts a local ontology to its original state, then executes the deletions that have occurred in both the local and global ontology, and than re-executes the changes that have been made to the local ontology. Further research about synchronization within a particular setting is required to find out whether these directions are sufficient.

6.5 Determining the Integrity of Mappings

A third application of the framework is determining the integrity of mappings between ontologies. That is, we consider the changes that occurred in a specific ontology and we determine whether mappings to that ontology are still usable after these changes. Because the validity of mappings can only be decided for a specific type of usage, we consider a particular context, namely subsumption reasoning in modular ontologies. In the next section, we present our approach to modular ontologies and we define what we mean with "integrity" of mappings. After that, we describe the actual method for verifying the mappings in Section 6.5.2.

6.5.1 Modular Ontologies

There are a number of reasons to mention why a modular setup is important to consider.

Distributed Systems: In highly distributed systems such as the Semantic Web, modularity naturally exists in terms of physical location. Providing interfaces and mechanisms for connecting these natural modules is a prerequisite for easy maintenance.

Large Ontologies: Modularization also helps to manage very large ontologies we find for example in medicine or biology. Here modularity helps to maintain and reuse

parts of the ontology as smaller modules are easier to handle than the complete ontology (Rector, 2003).

Efficient Reasoning: A specific problem that occurs in the case of distributed and large models is the problem of efficient reasoning. The introduction of modules with local semantics and clear interfaces will help to develop efficient reasoning methods (McIlraith and Amir, 2001).

In order to improve ontology maintenance and reasoning in the way suggested above, a modular ontology architecture should have the following characteristics.

Loose Coupling: In general, we cannot assume that two ontology modules have anything in common. This refers to the conceptualization as well as the specific logical language used for the interpretation of objects, classes or relations.

Self-Containment: In order to facilitate the reuse of individual modules we have to make sure that modules are self-contained. In particular, the result of certain reasoning tasks such as subsumption or query answering within a single module should be possible without having to access other modules.

Integrity: Having self-contained ontology modules may lead to inconsistencies that arise from changes in other ontology modules. We have to provide mechanisms for checking whether relevant knowledge in other systems has changed and for updating our modules accordingly.

Approach

Our approach can be summarized with the following three descriptions.

View-Based Mappings: We adopt the approach of view-based information integration. In particular, ontology modules are connected by conjunctive queries. This way of connecting modules is more expressive than simple one-to-one mappings between concept names but less expressive than the logical language used to describe concepts.

Compilation of Implied Knowledge: In order to make local reasoning independent from other modules, we use a knowledge compilation approach. The idea is to compute the result of each mapping query off-line and add the result as an axiom to the ontology module using the result. During reasoning, these axioms replace the query thus enabling local reasoning.

Change Detection and Automatic Update: Once a query has been compiled, the correctness of reasoning can only be guaranteed as long as the class hierarchy of the queried ontology module does not change. In order to decide whether the compiled axiom is still valid, we propose a change detection mechanism that is based on a taxonomy of ontological changes and their impact on the class hierarchy.

View-Based Mappings

Besides the concepts that are defined in a standard way, we consider *externally defined concepts* in the setting of modular ontologies. Externally defined concepts are assumed to be equivalent to the result of a query posed to another module in the modular ontology. This way of connecting modules is very much in spirit of view-based information integration which is a standard technique in the area of database systems (Halevy, 2001). We use the notion of an external concept definition which is an axiom of the form $C \equiv M : Q$ where M is a module and Q is a conjunctive query over the signature of M. Queries over ontological knowledge are defined as conjunctive queries, where the conjuncts are predicates that correspond to classes and relations of an ontology. Further, variables in a query may only be instantiated by constants that correspond to objects in that ontology. The claim that all conjuncts relate to elements of the ontology allows us to determine the answer to ontology-based queries in terms of instantiations of the query that are logical consequences of the knowledge base.

A model-based semantics for modular ontologies has been defined in Stuckenschmidt and Klein (2003), using the notion of a distributed interpretation proposed by (Borgida and Serafini, 2002) in the context of distributed description logics. Using the notion of logical consequence that can be defined on the basis of this semantics, we can turn our attention to the issue of reasoning in modular ontologies. For the sake of simplicity, we only consider the interaction between two modules in order to clarify the basic principles. Further, we assume that only one of the two modules contains externally defined concepts in terms of queries to the other module.

Compilation of Implied Knowledge

As mentioned in the requirements for modular ontologies, we are interested in the possibility of performing local reasoning. For the case of ontological reasoning, we focus on the task of deriving implied subsumption relations between concepts within a single module. For the case of internally defined concepts, this can be done using well established reasoning methods (Donini et al., 1996). Externally defined concepts, however, cause problems: being defined in terms of a query to the other module, a local reasoning procedure will often fail to recognize an implied subsumption relation between these concepts. Consequently, subsumption between externally defined concepts requires reasoning in the external module as the following theorem shows.

Theorem 1 (Implied Subsumption) *Let E_1 and E_2 be concepts that are externally defined by queries Q_1 and Q_2, then E_2 subsumes E_1 if and only if Q_2 subsumes Q_1 in the context of the ontology they refer to.*

This theorem which can easily be proven using the model-theoretic semantics of external concept definitions implies the necessity to decide subsumption between conjunctive queries in order to identify implied subsumption relations between externally defined concepts. In order to decide subsumption between queries, we translate them into internally defined concepts in the module they refer to. A corresponding sound and complete translation is described in (Horrocks and Tessaris, 2000). Using the resulting

concept definition, to which we refer as *query concepts*, we can decide subsumption between externally defined concepts by local reasoning in the external ontology.

We can avoid the need to perform reasoning in external modules each time we perform reasoning in a local module using the idea of knowledge compilation (Cadoli and Donini, 1997). The idea of compilation is to perform the external reasoning once and add the derived subsumption relations as axioms to the local module. These new axioms can then be used for reasoning instead of the external definitions of concepts. For the exact algorithm that is used to compile the new axioms, we refer to Stuckenschmidt and Klein (2003).

If we want to use the compiled axioms instead of external definitions, we have to make sure that this will not invalidate the correctness of reasoning results. We call this situation, where the compiled results are still a correct representation of as integrity. We formally define integrity as follows:

Definition 3 (Integrity) *We consider integrity of two ontology modules M, M_j to be present if $M, M_j \models M^c$ where M^c is the result of replacing the set of external concept definitions in M by compiled axioms.*

At the time of applying the compilation this is guaranteed by Theorem 1, however, integrity cannot be guaranteed over the complete life-cycle of the modular ontology. The problem is that changes to the external ontology module can invalidate the compiled subsumption relationships. In this case, we have to perform an update of the compiled knowledge.

6.5.2 Verifying Integrity

In principle, testing integrity might be very costly as it requires reasoning within the external ontology. In order to avoid this, we propose a heuristic change detection procedure that analyzes changes with respect to their impact on compiled subsumption relations. Work on determining the impact of changes on a whole ontology is reported in (Heflin and Hendler, 2000). As our goal is to determine whether changes in the external ontology invalidates compiled knowledge, we have to analyze the actual impact of changes on individual concept definitions. We want to classify these changes as either *harmless* or *harmful* with respect to compiled knowledge.

Determining Harmless Changes

As compiled knowledge reflects subsumption relations between query concepts, a harmless change is a set of modifications to an ontology that does not change these subsumption relations. Finding harmless changes is therefore a matter of deciding whether the modifications affect the subsumption relation between query concepts. We first look at the effect of a set of modifications on individual concepts.

Assuming that C represents the concept under consideration before and C' the concept after the change there are four ways in which the old version C may relate to the new version C':

1. the meaning of concept is not changed: $C \equiv C'$ (e.g. because the change was in another part of the ontology, or because it was only syntactical);
2. the meaning of a concept is changed in such a way that the concept becomes more general: $C \sqsubseteq C'$
3. the meaning of a concept is changed in such a way that the concept becomes more specific: $C' \sqsubseteq C$
4. the meaning of a concept is changed in such a way that there is no subsumption relationship between C and C'.

The same observations can be made for a relation before and after a change, denoted as R and R' respectively. The next question is how these different types of changes influence the interpretation of query concepts. We take advantage of the fact that there is a very tight relation between changes in concepts of the external ontology and implied changes to the query concepts using these concepts. A sketch of the proof for this lemma can be found in Stuckenschmidt and Klein (2003).

Lemma 1 (Monotonicity of Effect) *Let $c(Q)$ be the set of all concept names and $r(Q)$ the set of all relation names occurring in query Q, let further $C \in c(Q)$ and $R \in r(Q)$ then changing C has the same impact on the interpretation of Q as it has on the interpretation of C, in particular, we have $C \sqsubseteq C' \Longrightarrow Q \sqsubseteq Q'$ and $C' \sqsubseteq C \Longrightarrow Q' \sqsubseteq Q$ where Q' is the query as being interpreted after changing C. Analogously, a change of R has the same effect on the complete query.*

We can exploit this relation between the interpretation of concepts and queries in order to identify the effect of changes in the external ontology on the subsumption relations between different query concepts. First of all the above result directly generalizes to multiple changes with the same effect, i.e. a query Q becomes more general(specific) or stays the same if none of the elements in $c(Q) \cup r(Q)$ become more specific(general). Further, the subsumption relation between two query concepts does not change if the more general(specific) query becomes even more general(specific) or stays the same. Combining these two observations, we derive the following characterization of harmless change.

Theorem 2 (Harmless Change) *A change is harmless with respect to compiled knowledge (i.e. $Q_1 \sqsubseteq Q_2 \Longrightarrow Q'_1 \sqsubseteq Q'_2$) if for all compiled subsumption relations $C_1 \sqsubseteq C_2$ where C_i is defined by query Q_i we have:*

$$X' \sqsubseteq X \text{ for all } X \in c(Q_1) \cup r(Q_1)$$

$$X \sqsubseteq X' \text{ for all } X \in c(Q_2) \cup r(Q_2)$$

Proof 1 *We assume that $X' \sqsubseteq X$ for all $X \in c(Q_1) \cup r(Q_1)$. Applying lemma 1 with respect to all $X \in c(Q_1) \cup r(Q_1)$ we derive $Q'_1 \sqsubseteq Q_1$. We further assume that $X \sqsubseteq X'$ for all $X \in c(Q_2) \cup r(Q_2)$. Using lemma 1 we get $Q_2 \sqsubseteq Q'_2$. This leads us to $Q'_1 \sqsubseteq Q_1 \sqsubseteq Q_2 \sqsubseteq Q'_2$. Theorem 2 is established by transitivity of the subsumption relation.*

The theorem provides us with a correct but incomplete method for deciding whether a change is harmless. This basic method can be refined by analyzing the overlap of $c(Q_1)$ and $c(Q_2)$ in combination with the relations they restrict.

Characterizing Changes

Now we are able to determine the consequence of changes in the concept hierarchy on the integrity of the mapping, we still need to know what the effect of specific modifications on the interpretation of a concepts is (i.e. whether it becomes more general or more specific). As our goal is to determine the integrity of mappings without having to do classification, we describe what theoretically could happen to a concept as result of a modification in the ontology. To do this, we use the ontology of operations that is presented in Chapter 5. We use both basic changes but also complex changes, as they allow us to define the effect more precisely. Table 6.4 contains a list of operations and their effect on the classification of concepts.

Operation	**Effect**
Attach a relation to concept C	C: Specialized
Complex: Change the superclass of concept C to a concept lower in the hierarchy	C: Specialized
Complex: Restrict the range of a relation R *(effect on all C that have a restriction on R)*	R: Specialized, C: Specialized
Remove a superclass relation of a concept C	C: Generalized
Change the concept definition of C from primitive to defined	C: Generalized
Add a concept definition A	C: Unknown
Complex: Add a (not further specified) subclass A of C	C: No effect
Define a relation R as functional	R: Specialized

Table 6.4: Some modification to an ontology and their effects on the classification of concepts in the hierarchy.

The approach for characterizing changes is a heuristic, in the sense that the table specifies what *could* happen to concepts (i.e., the "worst-case" scenarios) and that for some operations the effect is "unknown" (i.e. unpredictable). In contrast to (Franconi et al., 2000) who provides complete semantics of changes we prefer to use heuristics in order to avoid expensive reasoning about the impact of changes.

Update Management

With the elements that we described in the section above, we now have a complete procedure to determine whether compiled knowledge in one module is still valid when other ontology modules have changed. The complete procedure is as follows:

1. create a list of concepts and relations that are part of the "subsuming" query of any compiled axiom;

2. create another list of concepts and relations that are part of the “subsumed” query of any compiled axiom;

3. achieve the modifications that are performed in the external ontology;

4. use the modifications to determine the effect on the interpretation of the concept and relations.

5. check whether there are concepts or relations in the first, “subsuming”, list that became more specific, or concepts or relations in the second, “subsumed”, list that became more general, or concepts or relations in any of the lists with an unknown effect; if not, the integrity of the mapping is preserved.

We describe the procedure in a more structured way in Algorithm 6.1. The algorithm

Require: Ontology Module M
Require: Ontology Module M_j
  for all compiled axioms $C_1 \sqsubseteq C_2$ in M^c **do**
    for all $X \in c(Q_1) \cup r(Q_1)$ **do**
      if effect on C is ’generalized’ or ’unknown’ **then**
        $M^c := Compile(M, M_j)$
      end if
    end for
    for all $X \in c(Q_2) \cup r(Q_2)$ **do**
      if effect on X is ’specialized’ or ’unknown’ **then**
        $M^c := Compile(M, M_j)$
      end if
    end for
  end for

Algorithm 6.1: Update

triggers a (re-)compilation step only if it is required in order to resume integrity. Otherwise no action is taken, because the previously compiled knowledge is still valid. In principle, all steps in the algorithm could be automated.

The approach described in this section provides us with a computationally cheap method to detect the impact of changes in ontologies on subsumption reasoning in a related ontology. In Chapter 8, a practical study is described in which the approach is executed.

6.6 Visualization

We can use complex operations to improve the user interface for the task of verifying and approving changes. Quite often, an ontology editor performs a number of changes that are all part of one “conceptual” operation. Some complex operations, like sibling move, capture this knowledge. Visualizing these operations could help the user to verify

modifications and to understand the potential effects of changes on applications that use the ontology.

We have developed a visualization for the difference between two ontology versions that explicitly shows some of the complex operations. The mechanism consists of displaying the new version of the ontology, and adding to it classes from the old version that were deleted or moved. Different visual clues, such as fonts, colors, and tooltips, are used to identify whether classes were added, deleted, moved to a new location, changed, and so on. A tool that implements this visualization is described in Chapter 7. Chapter 8 contains the result of a small-scale experiment that shows that there is some evidence to assume that the visualization indeed improves the understanding of changes in large ontologies.

6.7 Discussion

In this chapter, we described two things. First, we pictured the general process of using the change framework, and second, we explained how the change framework can be used in three different ontology tasks for which evolution is relevant. The selection of tasks is a more-or-less arbitrary choice between possible tasks. There are many other tasks imaginable that also could be linked to the change framework.

For example, a mechanism could be developed to keep an ontology in a consistent state, or within a specific OWL variant. To implement this, a set of rules—in the same flavor as the ones described in (Stojanovic et al., 2002)—has to be developed that specify required follow-up changes for specific change operations.

Also, we can use the compatibility effects of the ontology changes (see Section 6.2) to decide whether the identity of an ontology has changed. If so, a new identifier has to be assigned.

In the next two chapters, we will apply some of the methods described in this chapter. Chapter 7 describes three computerized tools that implement (parts of) the processes, whereas Chapter 8 contains three practical studies in which we used the tools and methods.

Part III

Applying the framework

Chapter 7

Tool Support

***Note**: Parts of this chapter are based on earlier publications. Large chunks of Section 7.1 have been published in (Klein et al., 2002a), co-authored by Dieter Fensel, Atanas Kiryakov and Damyan Ognyanov. The OntoView tool is jointly developed with Atanas Kiryakov and Damyan Ognyanov.*
The tool described in Section 7.3 is developed by Natasha Noy, based on joint work on complex change operations. It has been presented as poster at ISWC 2003 (Noy and Klein, 2003). Follow-up work as been published at ISWC 2004 (Noy et al., 2004).

In this chapter and the following one, we apply the framework described in the last three chapters. As the framework gives directions for coping with a problem that will only exist in a future situation, we cannot perform realistic evaluations of the described framework and the methods. Instead, we will do two things to provide some evidence for the framework. In this chapter we will describe three computerized tools that can automate some of the tasks that play a role in the framework. In the next chapter, we describe three practical studies in which we illustrate how to use parts of the framework with realistic ontologies.

We first describe the tool *OntoView*, which implements a change detection procedure for RDF-based ontologies. The role of this tool is to produce a transformation set (as is described in Section 6.2.1). Then, we describe two extensions to the PROMPTdiff tool (Noy and Musen, 2002). The first extension uses the mappings produced by PROMPTdiff as a basis for producing a transformation set. The second extension is able to detect some composite changes and presents these in a conveniently arranged way to the user.

7.1 Change detection in RDF-based ontologies

In this section, we describe a system called OntoView. Its main function is to compare versions of ontologies and to highlight the differences. OntoView is inspired by the Concurrent Versioning System CVS (Berliner, 1990), which is used in software development

to allow collaborative development of source code. The first implementation was also based on CVS and its web-interface CVSWeb[1]. However, during the ongoing development of the system, we gradually shifted to a complete new implementation that is build on available components, such as the Jena Semantic Web toolkit[2] and an ontology storage system, i.e. Sesame[3].

The underlying ideas of the system do not depend on a specific ontology language. However, the implementation of specific parts of the system assume RDF based languages. For example, this is the case with the mechanism to detect changes. In the remainder of this article, we will use DAML+OIL[4] (Fensel et al., 2000; Fensel and Musen, 2001) and RDF Schema (RDFS) (Brickley and Guha, 2000) as ontology languages.

The main feature of OntoView is its ability to compare ontologies at a structural level. The comparison function is inspired by UNIX `diff`, but the implementation is quite different. Standard `diff` compares file version at line-level or at character-level, highlighting the lines that textually differ in two versions. OntoView, in contrast, compares version of ontologies at a *structural* level, showing which definitions of ontological concepts or properties have changed. An example of the output of the tool after comparing two versions of an ontology represented in DAML+OIL is shown in Figure 7.1.[5]

The comparison function distinguishes between additions, deletions and definition changes. The order of the definitions in the file and the particular RDF representation is not important. Each type of change is highlighted in a different color.

7.1.1 Detecting changes

There are two main problems with the detection of changes in ontologies. The first problem is the abstraction level at which changes should be detected. Abstraction is necessary to distinguish between changes in the representation that affect the meaning, and those that don't influence the meaning. It is often possible to represent the same ontological definition in different ways. For example, in RDF Schema, there are several ways to define a class:

```
<rdfs:Class rdf:ID="ExampleClass"/>
```

or:

```
<rdf:Description rdf:ID="ExampleClass">
  <rdf:type rdf:resource="...org/2000/01/rdf-schema#Class"/>
</rdf:Description>
```

Both are valid ways to define a class and have exactly the same meaning. Such a change in the representation would not change the ontology. Thus, detecting changes in the *representation* alone is not sufficient.

[1] Available from `https://www.cvshome.org/cyclic/cyclic-pages/web-cvsweb.html`

[2] See `http://www.hpl.hp.com/semweb/jena.htm`

[3] A demo is available at `http://sesame.aidministrator.nl`

[4] Available from `http://www.daml.org/language/`

[5] This example is based on fictive changes to the DAML+OIL example ontology, available from `http://www.daml.org/2001/03/daml+oil-ex.daml`.

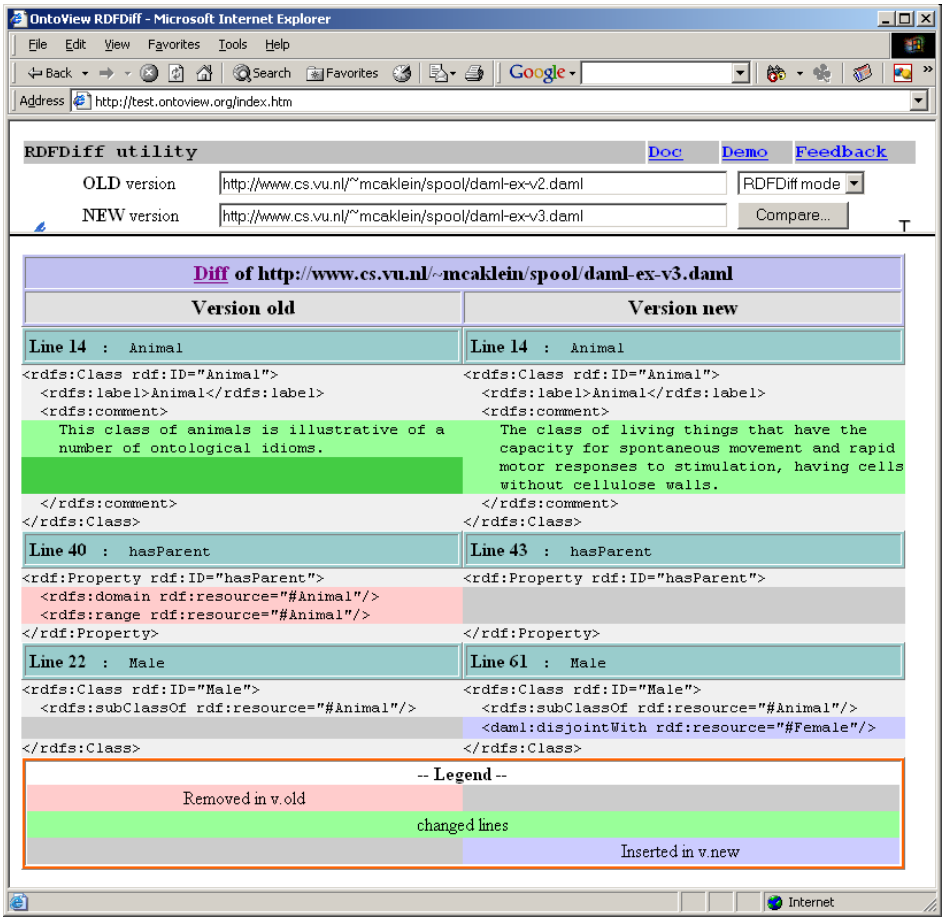


Figure 7.1: Comparing two ontologies

However abstracting too far can also be a problem: considering the *logical meaning* only is not enough. In (Bechhofer et al., 2001) is shown that different sets of ontological definitions can yield the same set of logical axioms. Although the logical meaning is not changed in such cases, the ontology definitely is. Finding the right level of abstraction is thus important.

Second, even when we have found the correct level of abstraction for change detection, the conceptual implication of a change is not yet clear. Because of the difference between conceptual changes and explication changes (as described in section 4.1.1), it is not possible to derive the conceptual consequence of a change completely from the visible change (i.e., the changes in the definitions of concepts and properties). Heuristics can be used to suggest conceptual consequences, but the intention of the engineer determines the actual conceptual relation between versions of concepts. In the next section,

we explain the algorithm that we used to compare ontologies at the correct abstraction level.

7.1.2 Rules for changes

The algorithm uses the fact that the RDF data model (Lassila and Swick, 1999) underlies a number of popular ontology languages, including RDF Schema and DAML+OIL. The RDF data model basically consists of triples of the form `<subject, predicate, object>`, which can be linked by using the object of one triple as the subject of another. There are several syntaxes available for RDF statements, but they all boil down to the same data model. A set of related RDF statements can be represented as a graph with nodes and edges. For example, consider the following DAML+OIL definition of a class "Person".

```
<daml:Class rdf:ID="Person">
  <rdfs:subClassOf rdf:resource="#Animal"/>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasParent"/>
      <daml:toClass rdf:resource="#Person"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

When interpreted as a DAML+OIL definition, it states that a "Person" is a kind of "Animal" and that the instances of its "hasParent" relation should be of type "Person". However, for our algorithm, we are first of all interested in the RDF interpretation of it. That is, we only look at the triples that are specified, ignoring the DAML+OIL meaning of the statements. Interpreted as RDF, the above definition results in the following set of triples:

subject	**predicate**	**object**
Person	rdf:type	daml:Class
Person	rdfs:subClassOf	Animal
Person	rdfs:subClassOf	*anon-resource*
anon-resource	rdf:type	daml:Restriction
anon-resource	daml:onProperty	hasParent
anon-resource	daml:toClass	Person

This triple set is depicted as a graph in Figure 7.2. In this figure, the nodes are resources that function as subject or object of statements, whereas the arrows represent properties.

The algorithm that we developed to detect changes is the following. We first split the document at the first level of the XML document. This groups the statements by their intended "definition". The definitions are then parsed into RDF triples, which results in a set of small graphs. Each of these graphs represent a specific definition of a concept or a property, and each graph can be identified with the identifier of the concept or the property that it represents.

Then, we locate for each graph in the new version the corresponding graph in the previous version of the ontology. Those sets of graphs are then checked according to a

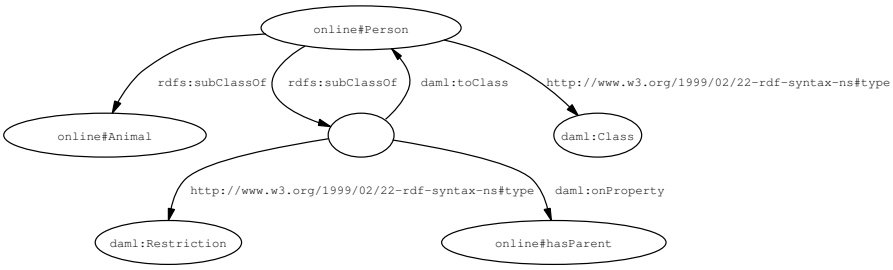


Figure 7.2: An RDF graph of a DAML class definition.

number of rules. Those rules specify the "required" changes in the triples set (i.e., the graph) for a specific change operation.

The rules have the following format:

```
IF exist:old
     <A, Y, Z>*
   exist:new
     <X, Y, Z>*
   not-exist:new
     <X, Y, Z>*
THEN change-type A
```

They specify a set of triples that should exists in one specific version, and a set that should not exists in another version (or the other way around) to signal a specific type of change. With this rule mechanism, we were able to specify almost all types of change (except for the identifier change).

For example, a rule to specify a change in the property type looks as follows:

```
IF exist:old
     <X, rdf:type, rdf:#Property>
     <X, rdf:type, daml:#TransitiveProperty>
   exist:new
     <X, rdf:type, rdf:#Property>
   not-exist:new
     <X, rdf:type, daml:#TransitiveProperty>
THEN Unset_Transitivity
```

The rules are specific for a particular RDF-based ontology language (in this case DAML+OIL), because they encode the interpretation of the semantics of the language for which they are intended. For another language the rules would have specified other combinations of changes that signal a higher-level change. The semantics of the language are thus encoded in the rules. For example, the rules above do not point out a change in values of predicates, but a change in the type of property. This is a change that is related to the specific semantics of DAML+OIL.

The described mechanism relies on the "materialization" of all `rdf:type` statements that are encoded in the ontology. In other words, the closure of the RDF triples according

to the used ontology language has to be computed. For this materialization, the entailment and closure rules in the RDF Model Theory[6] can be used. For example, the rules in example above depend on the existence of a statement `<X,rdf:type,rdf:#Property>`. However, this statement can only be derived using the semantics of the `rdfs:subPropertyOf` statement, which—informally spoken—says that if a property is an instance of type X, then it is also an instance of the supertypes of X. The application of the rules thus has to be preceded by the materialization of the superclass- and superproperty hierarchies in the ontology.

7.1.3 Discussion

The tool detects changes in RDF-based ontologies and uses rules to find specific operations. Doing this, it comes up with a set of change operations between ontology versions. This set can be used as input for other processes that are described in Chapter 6.

The change detection methods that we developed follows the principle of abstracting sets of smaller changes into higher level changes. The algorithm that we developed starts with an ontology that is represented in the RDF data model. It first parses a textual representation of the ontology into RDF triples, in order to find the changes in the data model instead of the textual representation, and search for added and deleted statements. Then, it groups the statements into individual class- and property definitions of the ontology. The changes in the sets of statements that form these definitions are then analyzed to detect the basic changes from our change ontology. Further, the basic changes are aggregated into complex changes. Each step in this procedure results in a higher level representation of the differences, and allow us the derive other conclusions about the consequences of the change.

A useful extension of the tool would be to allow humans to *characterize* the conceptual implication of the changes between two versions of the definitions that are shown. The user could be given the option to label changed versions of concepts either as "identical" (i.e., the change is an explication change), or as "conceptual change", e.g. using a drop-down list next to the definition. In the latter case, the user could specify the conceptual relation between the two versions of the concept. For example, the change in the definition of "hasParent" could by characterized with the relation $\texttt{hasParent}_{1.1}$ $\texttt{subPropertyOf}$ $\texttt{hasParent}_{1.3}$. This would give the tool an additional function in the framework, namely recording the conceptual relation between concepts or properties.

7.2 Change Operations in PROMPTdiff

PROMPTdiff is a tool developed by Noy and Musen (2002), as a plug-in for the Protégé ontology editor.[7] PROMPTdiff uses a set of heuristics to find mappings between frames (i.e. concepts, slots, etc.) in the old version of the ontology and frames in the new version. In this section, we describe how we extended the functionality of this tool and integrated it with our framework.

[6] `http://www.w3.org/TR/rdf-mt/`

[7] Both the editor and the plug-in can be achieved via `http://protege.stanford.edu`.

7.2.1 Basic Functionality

The PROMPTdiff tool takes two Protégé projects (consisting of an ontology and instance data) and tries to match frames in the old version with frames in the new version. It uses several heuristics that have proven to be useful for general ontology alignment. These heuristics are applied with less strict assumptions about the amount of similarity between concepts that is required to form a match. This is possible because the probability that two concept descriptions refer to the same concepts is much higher when one ontology is derived from another than when they are developed on their own.

Example of the heuristics used are "if a frame is of the same type (i.e. class, slot, instance) and has the same name, it is probably the same", and "if there is a matched class which has only one unmatched subclass in both versions, those two subclasses are probably the same". The resulting pairs, i.e. the frames that are matched across versions, are called "images" of each other.

The matched frames are classified into three groups, i.e. *unchanged*, *isomorphic* and *changed*. These levels indicate whether the matching frames are different enough from each other to warrant the user's attention. If a frame and its image are marked as "unchanged", they have the same slots with the same values. In other words, they have the same set of relations and none of the related frames have changed. An "isomorphic" change implies that the corresponding slots and facet values of two frames are images of each other, but not necessarily identical images. In informal terms, this means that the two frames have the same set of relations, but that related frames might have changed. Finally, "changed" means that the frames have slots or facet values that are not images of each other. Informally, they have a different set of relations to other frames. For "changed" and "isomorphic" images, the tool also shows an explanation why the image is of a particular type. For example, it lists the slot that is new or the value that has changed.

Figure 7.3 on the following page shows a screenshot of the result of running the PROMPTdiff tool. The ontologies that are used are the parts of the UNSPSC classification hierarchy. The practical study in Section 8.3 contains a more extensive description of this ontology. The main element in the screen is an image table that shows mappings between frames. The first two rows are added frames that do not have a image in the original version. Similarly, the next seven lines represent deletions. The following five lines depict frames that are mapped onto each other, but with different names. The last column explains the reason behind the mapping. For example, for the selected row the reason is that both versions of the class have the same superclass and subclasses. The column "map level" shows in which category the change is classified. This is illustrated in the bottom part of the screen, where the affected relations to other frames are shown. For example, the change in the selected row is categorized as "isomorphic", which means that it has the same slots and values but that at least one of the referenced frames. In this case its superclass ("Electical_equipment_and_components....") has been modified.

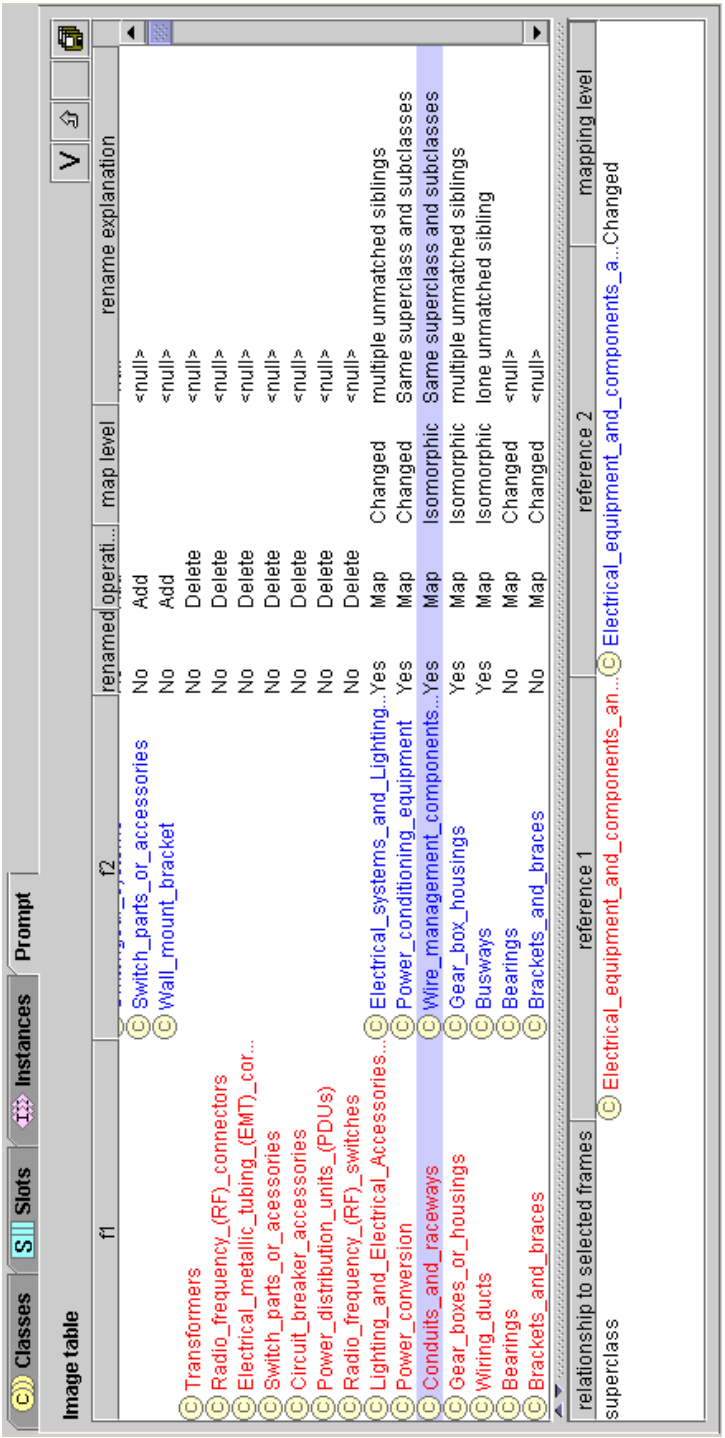


Figure 7.3: A screenshot of the original version of PROMPTdiff.

7.2.2 Place within Framework

When we relate this to our framework, we see that PROMPTdiff produces the *evolution relation* between the elements of two ontology versions. That is, it specifies which frame has likely evolved into which other frame. As explained in the processes in Chapter 6, the evolution relation provides a basis for producing most other forms of change information.

In principle, the mappings produced by PROMPTdiff could be exported in our change representation language as "unspecified" changes, i.e. changes for which only the "to" and "from" are specified. For the example in Figure 7.3 on the preceding page, this would look as follows. Note that a change specification uses the assumption that if two frames have the same local part of their identifier (in this case their name), they are mapped by default. Consequently, only for the frames that have a different name an explicit change is specified.

```
<ov:Change_Specification>
  <ov:source rdf:resource="http://www.eccma.com/unspsc/8/0"/>
  <ov:target rdf:resource="http://www.eccma.com/unspsc/8/4"/>
</ov:Change_Specification>

...

<ov:Change>
  <ov:from rdf:resource="&old;Gear_boxes_or_housings"/>
  <ov:to rdf:resource="&new;Gear_box_housings"/>
</ov:Change>

<ov:Change>
  <ov:from rdf:resource="&old;Wiring_ducts"/>
  <ov:to rdf:resource="&new;Busways"/>
</ov:Change>
```

The exported mapping can then be validated by a human, or directly be used as input—together with other sources—for one of the other processes, e.g. for suggestion conceptual relations, or for detecting complex changes.

Instead of exporting the mappings directly, we have extended the tool to perform two other tasks. First, it uses that mappings together with the ontology versions to create a set of change operations that translates the original version into the newer version, and second, it analyses the ontology versions and finds a number of rich operations.

7.2.3 Producing Transformations

To create a minimal transformation set, we extended the program in such a way that it compares the complete definitions of the mapped frames. For each of these frame pairs, the program looks at its slots, their values, their facets and the values of the facets. If there is a difference for one of these items, the slot–facet–value triples of both versions of the frames are stored. This results in change information such as is sketched Figure 7.4 on the following page.

To get change operations, this "raw" change information has to be translated into higher-level information. This involves two steps. First, changes in the built-in slots and facets of the OKBC knowledge model are interpreted. For example, a change in the slot

```
Class "Merlot"
 OLD:
   slot : :DIRECT-SUPERCLASSES
   facet: -
   value: "White Wine"
 NEW:
   slot : :DIRECT-SUPERCLASSES
   facet: -
   value: "Red Wine"

Class "Red Wine"
 OLD:
   slot : :DIRECT-SUBCLASSES
   facet: -
   value: -
 NEW:
   slot : :DIRECT-SUBCLASSES
   facet: -
   value: "Merlot"
```

Figure 7.4: An example of the "raw" change information for two versions of two classes.

`:DIRECT-SUBCLASSES` is interpreted as a change in the "subclass" relation, and a change in the facet `:VALUE-TYPE` of a slot is interpreted as a change of a "$\forall$-slot-restriction". This interpretation allows us to distinguish e.g. between an addition of a slot-restriction and the addition of superclass-relations. The second step is the aggregation of successive deletions and additions on the same slot or facet into "modification" operations. The result of the interpretation is a list of change operations for each frame with—when appropriate—the old and new values.

The lists of operations contain the changes for every relation between a frame and an other frame. This causes two problems. The first problem is that the aggregated operation set is redundant and that it therefore does not form a *minimal* transformation set. For the example in Figure 7.4, we would have both an operation for the change of the superclass relation for the frame "Merlot" and an operation for the change of the subclass relation for the frame "Red Wine". Another problem with the change list is that it contains change operations for elements that are not defined *within* the frame, but inside other frames. For example, the `:DIRECT-INSTANCES` slot contains the instances of a class, but this is defined within the instance itself, by declaring its "type".

To solve this problem, we have extend the PROMPTdiff classification of changes with two specific types of changes, namely "direct-change" and "implicit-change". This results in five different categories for the *extent* of the change.

Implicitly-changed: the slots or facets that are defined within the frames are different or have a different frame as their value, but all these differences are the consequence of the direct-change of another frame (i.e. a deletion). For example, if a slot is removed from an ontology, the slot-restrictions for that slot are also removed from the frames. The latter removals are implicit changes.

level	operation	slot	facet	reference 1	reference 2
Change (direct)	slot restriction filler changed to subclass	S Beginning		C Point_in_Time	C Y-M-D-H-M-S_Point_in_Time
Change (direct)	slot restriction filler changed to subclass	S Ending		C Point_in_Time	C Y-M-D-H-M-S_Point_in_Time
Isomorphic	template slot altered			S Beginning	S Beginning
Isomorphic	template slot altered			S Ending	S Ending

Figure 7.5: A number of change operations detected by the PROMPTdiff extension for a class called "Interval_of_Time".

Directly-changed: the slots or facets that are defined within the frames are different or have a different frame as their value. For example, the change of the cardinality (a facet) of a slot-restriction.

Changed: the slots or facets of the frames are different or have a different frame as their value. An example is the introduction of a subclass with another class; this is defined within the subclass itself, and is therefore not a direct change.

Isomorphic: the sets of slots or facets of the frames are equal but at least one of them has a directly-changed frame as its value. For example, if the class that forms the range of a slot is changed, the slot itself is called "isomorphic".

Unchanged: the sets of slots or facets of the frames are equal but at least one of them has a changed frame as its value.

Note that the set of implicitly-changed frames is a subset of the set of directly-changed frames, which is again a subset of the changed frames. The sets of isomorphic and unchanged frames are disjoint with each other and with the set of changed frames.

The direct changes are the ones that form the minimal transformation set. We provide an export function that saves the list of operations in a file. The category of implicit changes is useful to filter out the changes that were not the result of editing action within a frame. For example, if for validation purposes one wants to show all editing actions that have occurred between two versions, one should select the direct-changes without the implicit changes. However, in a transformation set the implicit changes *are* required, as they are necessary for reversing the change.

We also implemented some algorithms in the tool to detect some rich operations, especially the ones that specify that a specific filler became a superset or a subset. For this, we queried the knowledge model of the old versions of the ontology to know whether the image of a new value was a superclass or subclass of the old value. If this was the case, we changed the operations accordingly. For example, the operation Modify_Superclass could be specialized into Modify_Superclass_To_Subclass, or Modify_Range into Modify_Range_to_Subrange. Figure 7.5 illustrates how the tool shows the detected changes.

7.3 Visualizing changes

We will now describe the PROMPTdiff user interface for visualizing some of the complex changes between ontology versions. The interface was inspired by the interface that Microsoft Word uses to present changes. In Word, the text that is deleted is crossed out and the added text is underlined, and there is also color coding for these two types of changes. Note however, that Word does not identify complex operations, such as move.

7.3.1 Visual metaphors

We use a similar user-interface paradigm to show differences between ontology versions. However, we show *structural* differences rather than text differences and we identify and visualize *complex changes*, such as the ones identified in Chapter 5. Figure 7.6 on the next page shows how PROMPTdiff presents the result of comparing two versions of the UNSPSC ontology. The UNSPSC ontology[8] is a standardized hierarchy of products and services that enables users to consistently classify the products and services they buy and sell. Based on the input of the users, a new version is published from time to time. Typical differences between two versions are additions of new products, or re-classifications of existing products.

In the figure showing the PROMPTdiff result, the classes that were deleted are crossed out, the added classes are underlined, and classes that were renamed or changed are in bold. We also use color coding to make the changes even more apparent. So, for example, we can see that the classification of "Power_conditioning_equipment" has undergone a number of changes: three classes were added ("Distribution_Power_Transformers", "Instrument_Transformers", and "Power_Supply_Transformers"), one was deleted ("Transformers"), and one was renamed ("Power_distribution_units_PDUs"). The tooltip shows the old name of the renamed class.

Figure 7.7 on page 140 shows complex changes in these two versions of the UNSPSC ontology: The addition of several classes rooted at "Distribution_and_Control_centers_and_accessories" as an addition of a tree of classes. The class "Electrical_equipment_and_components_and_supplies" was moved to this location from a different position in the tree. The icon at the root of the added subtree has an overlayed add icon (⊕) indicating that all classes in this subtree have the same status—they were all added in this version. Similarly, if a whole tree was deleted, an overlayed delete icon (⊗) identified the tree-level operation. The tooltip at the moved class indicates where the class was moved from.

Figure 7.8 on page 140 shows the moved class in its old position in the hierarchy: the class appears in grey and the tooltip indicates where the class was moved to.

In addition, a warning icon (⚠) overlayed with the class icon indicates that the subtree rooted at the class has undergone some changes (Figure 7.7 on page 140). However, unlike with the tree add (when we use the ⊕ icon), the subtree either contains different changes or has both changed and unchanged classes. In Figure 7.7 on page 140, for example, the user can see that one of the subtrees rooted at a subclass of "Electrical_

[8] See `http://www.eccma.org/unspsc/`.

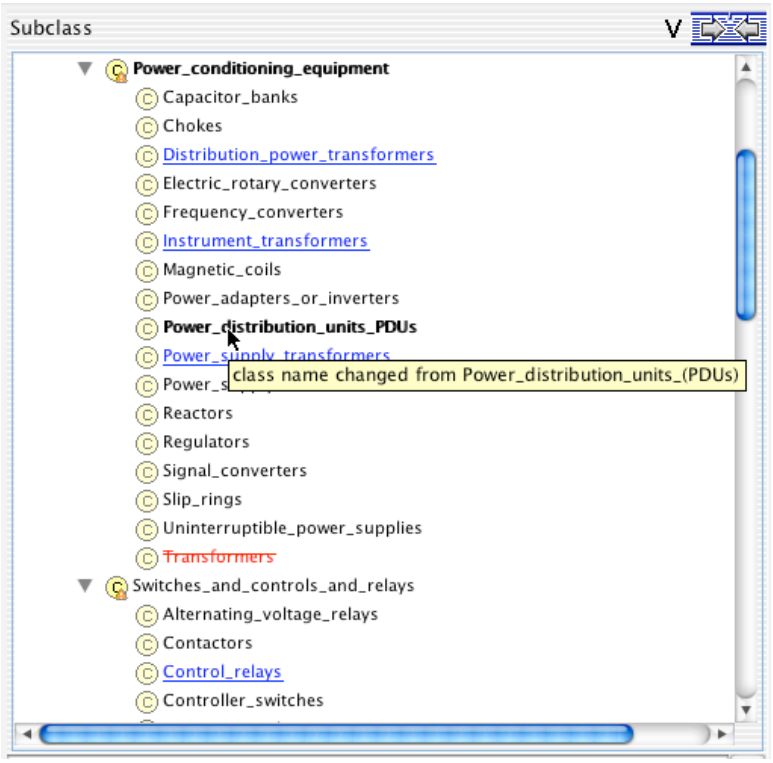


Figure 7.6: Comparison of two versions of the UNSPSC ontology in PROMPTdiff. The classes that were deleted are crossed out and the added classes are underlined.

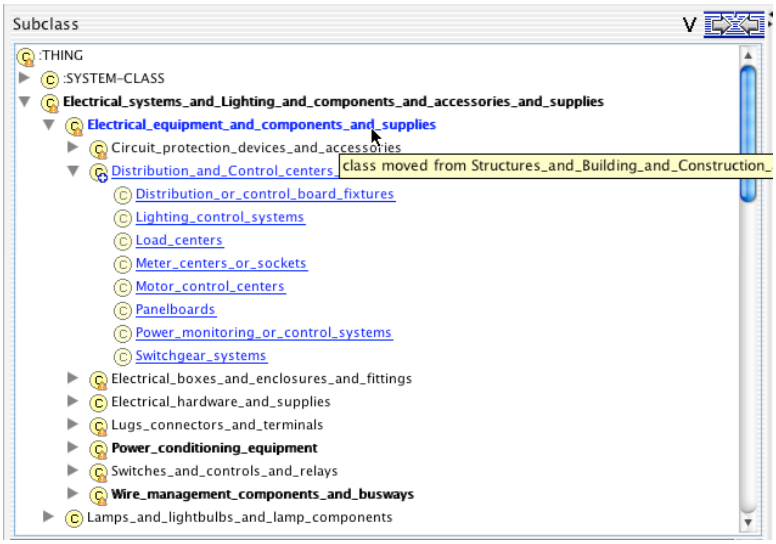


Figure 7.7: Comparison of two versions of the UNSPSC ontology in PROMPTdiff. The added classes are underlined. The class icon with an add icon indicates an addition of the whole tree. The class icon with an overlayed warning icon indicates that the subtree rooted at the class has undergone some changes.

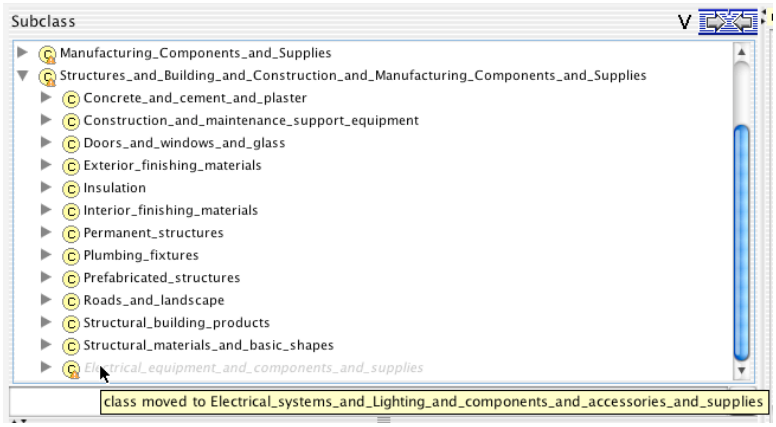


Figure 7.8: Comparison of two versions of the UNSPSC ontology in PROMPTdiff. The class "Electrical_equipment_and_components_and_supplies" was moved to a different place in the class hierarchy.

equipment_and_components_and_supplies" (i.e., "Electrical_hardware_and_supplies" has not changed at all, whereas all the others have changed.

To summarize, we visualize two types of changes: (1) class-level changes and (2) tree-level changes. For class-level changes, the class-name appearance indicates whether the class was added, deleted, moved to a new location, moved from a different location, or its name or definition has changed. If all classes in a subtree have changed in the same way (were all added or deleted, for example), then the changed icon at the subtree root indicates that the tree-level operation.

7.3.2 Navigation among changes

In practice only a small portion of an ontology changes from one version to another (Noy and Musen, 2002). Furthermore, many ontologies have deep class hierarchies with many levels and, since there are simply more classes at the lower levels, changes are more likely to occur at those levels. Therefore, these changes are not directly visible if a user expands only the first few levels of a class-hierarchy tree. Therefore, just as users can navigate to the "next" and "previous" changes in comparing text documents, we enable PROMPTdiff users to iterate through changes using the "next" and "previous" buttons. However, unlike text documents, tree hierarchies are not linear. Therefore, the notion of *next* and *previous* node in the tree is not well defined.

For a tree node N, we define the **next node** as the next node in a depth-first traversal of the tree with backtracking. For a tree node N, we define the **previous node** as a reversal of the next-node operation: Node P is the previous node for N, $previous(N)$, if N is equal to $next(P)$.

For example, in Figure 7.6 if class "Power_conditioning_equipment" is selected, the next changed class is "Distribution_power_transformers". If "Slip_rings" is selected, the next change is "Transformers" and the next one is "Control_relays". Note that "Control_relays" is in the next subtree.

7.3.3 Conclusions and Future Work

In this section, we have presented a tool for examining changes between ontology versions. The PROMPTdiff-extension displays the new version of the ontology, adding to it classes from the old version that were deleted or moved. It uses different visual clues, such as fonts, colors, and tooltips, to identify whether classes were added, deleted, moved to a new location, changed, and so on. We identify and present to the user both basic and complex changes. Currently, PROMPTdiff does not display all the changes that we presented in Chapter 5, although internally it identifies more of them.

Another natural extension of the current tool would be enabling users to accept and reject changes. For example, a user can reject a change that added a new class, and the system will delete the class from the new version. If the user accepts this change, the visualization for the class will change to the default one (it will no longer stand out as one of the changes).

Chapter 8

Practical Studies

***Note**: The study in Section 8.2 is published as part of a paper in the ISWC 2003 workshop on Semantic Integration (Klein and Stuckenschmidt, 2003).*

This chapter describes three studies in which we show how some of the elements of the framework can function in realistic settings. First, we study the changes within a line of ontology versions. We use our change specification language (Chapter 5) to represent one specific version change, and we show how we can use this specification together with the ontology of change operations to query for addition information about the change.

In a second study, we show how the process to determine the integrity of mappings between ontologies (as is described in Section 6.5) works when applied to an (artificial) ontology that contains mappings to a real-world evolving ontology. For this study, we use an ontology that evolved within a case study on a methodology for ontology development.

Finally, we describe an experiment in which we compare an "intelligent" visualization of changes (as is described in Chapter 7) with a traditional visualization. As subject ontologies, we used two versions of the UNSPSC ontology.

8.1 Specifying and Querying a Change Specification

In this study, we will show that our change specification language can be used to specify changes in a realistic ontology evolution scenario. For this, we use an evolution line of the BioSAIL ontology.[1]

We start a description of the ontology itself, followed by a discussion of its evolution. Then, in Section 8.1.3, we specify the change using the language that has been proposed in Chapter 5. After that, we illustrate how we can query the specification. Appendix C

[1]We would like to thank Zachary Pincus for providing us with the complete trace of ontology versions together with an explanation of the rationale of the changes, and for the description of its development and usage.

gives the complete specification of the changes, the queries and the answers to the queries that are discussed in this section.

8.1.1 BioSAIL Ontology

The BioSAIL ontology is developed within the BioSTORM project (Buckeridge et al., 2002). The general goal of this project is to develop and evaluate knowledge representations and problem solving methods to facilitate public health surveillance of multiple disparate data sources.[2]

Data for monitoring the health of the population are becoming available from many different sources. More-and-more, surveillance systems are exploring the analysis of *non-clinical* health data, such as school absenteeism and pharmacy sales, in an attempt to increase the timeliness of outbreak detection. However, it is difficult to use this data directly, as there are no standards for the encoding of this data.

In order to integrate these data sources, the BioSAIL (BioSTORM Systems Abstraction and Interface Layer) ontology is developed. BioSAIL describes basic conceptual elements of non-clinical data and allows users to build detailed, customized descriptions of specific data sources and formats from these elements.

The versions of the BioSAIL ontology that we use contain on average 180 classes and around 70 properties. The ontology is modeled with the Protégé editor and consequently uses the Protégé knowledge model (Fergerson et al., 2000). This knowledge model is a slightly adapted version of the OKBC knowledge model (the latter one is described in Section 5.2).

The main concepts that are modeled in the BioSAIL ontology are “data providers” and “measurements”. Measurements are described with a “measurement specification”, using “LOINC terms”. The relations between these concepts are depicted in Figure 8.1.

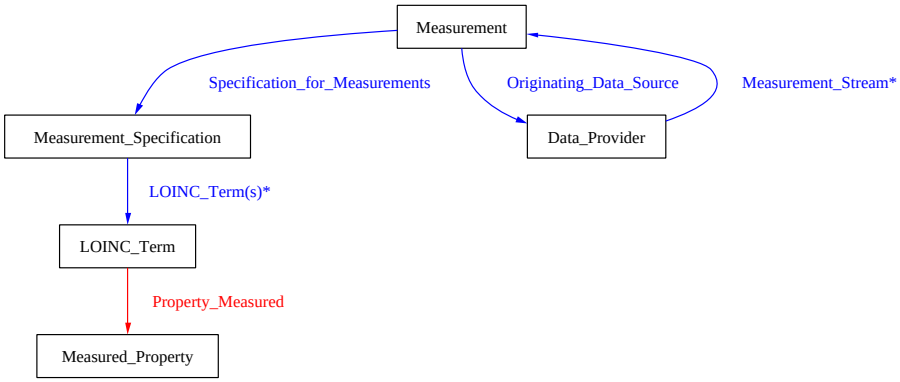


Figure 8.1: The main concepts in the BioSAIL ontology and their relations.

[2] `http://smi-web.stanford.edu/projects/biostorm/`

Example types of *data providers*, are “schools”, “employers”, “pharmacies” and “emergency call centers”. *Measurements* are pieces of data or groups of related data. A *measurement specification* describes a measurement by providing metadata about its elements. For example, a measurement could be about the “demographics” in a specific hospital. The measurement specification would say that it consists of the elements “gender” and “birth-date”. The names of these elements are taken from a standardized vocabulary, called LOINC. This stands for *Logical Observation Identifiers Names and Codes* and it provides universal identifiers for laboratory and other clinical observations.[3] In the BioSAIL ontology, the LOINC terms also link to a *measured property*, which is a controlled vocabulary of *types* of properties. Figure 8.2 shows a part of the hierarchy of these properties.

Besides these concepts, the ontology also contains utility classes, with hierarchies of concepts to describe time points and time intervals, locations, the dimension of measurements, like “mass”, “volume”, “area” or “frequency”, etcetera.

8.1.2 Ontology Evolution

The change of ontology versions that we have range from version 1.6 release 1 to version 2.2 release 1, in total 16 different versions. According to the author of the ontology, most changes are minor: moving of classes in the hierarchy, adding and subtracting slots, changing slot overrides, etc. There is a small number of major changes, for example the change of groups of attributes to meta slots. Table 8.1 summarizes the changes between the different versions.

Version		Classes				Slots				Instances				
from	to	A	D	R	C	A	D	R	C	A	D	R	C	Total
1.6r1	1.6r2	–	–	–	–	–	–	–	–	–	–	–	3	3
1.6r2	1.6r3	–	–	1	2	2	2	–	2	–	–	–	13	22
1.6r3	1.6r4	–	–	1	2	1	–	–	–	–	–	–	–	4
1.6r4	1.6r5	–	–	–	–	–	–	–	–	–	–	–	–	0
1.6r5	2.0r1	–	–	1	2	–	1	–	–	–	–	–	–	4
2.0r1	2.0r2	33	1	2	13	7	8	1	2	17	–	–	13	97
2.0r2	2.0r3	6	7	4	15	1	–	–	4	1	–	–	8	46
2.0r3	2.0r4	–	1	–	–	–	–	–	–	–	–	–	–	1
2.0r4	2.1r1	21	24	2	1	3	2	–	9	–	25	–	–	87
2.1r1	2.1r2	7	–	5	19	3	–	–	4	3	6	–	21	68
2.1r2	2.1r3	–	–	–	3	–	–	–	1	–	–	–	8	12
2.1r3	2.1r4	7	–	2	3	2	–	–	4	9	–	–	21	48
2.1r4	2.1r5	1	–	–	2	–	–	5	–	1	–	–	1	10
2.1r5	2.1r6	–	–	–	–	–	–	–	–	–	–	–	1	1
2.1r6	2.2r1	–	–	–	7	–	–	–	3	192	–	–	3	205
Total		75	33	18	69	19	13	6	29	223	31	0	92	608

Table 8.1: Changes in the BioSAIL ontology. Each row represents the changes from the version number in the first column to the next version. ‘A’ stands for ‘addition’, ‘D’ for ‘deletion’, ‘R’ for ‘renaming’, and ‘C’ for ‘change in the definition’.

[3] See `http://www.loinc.org/`

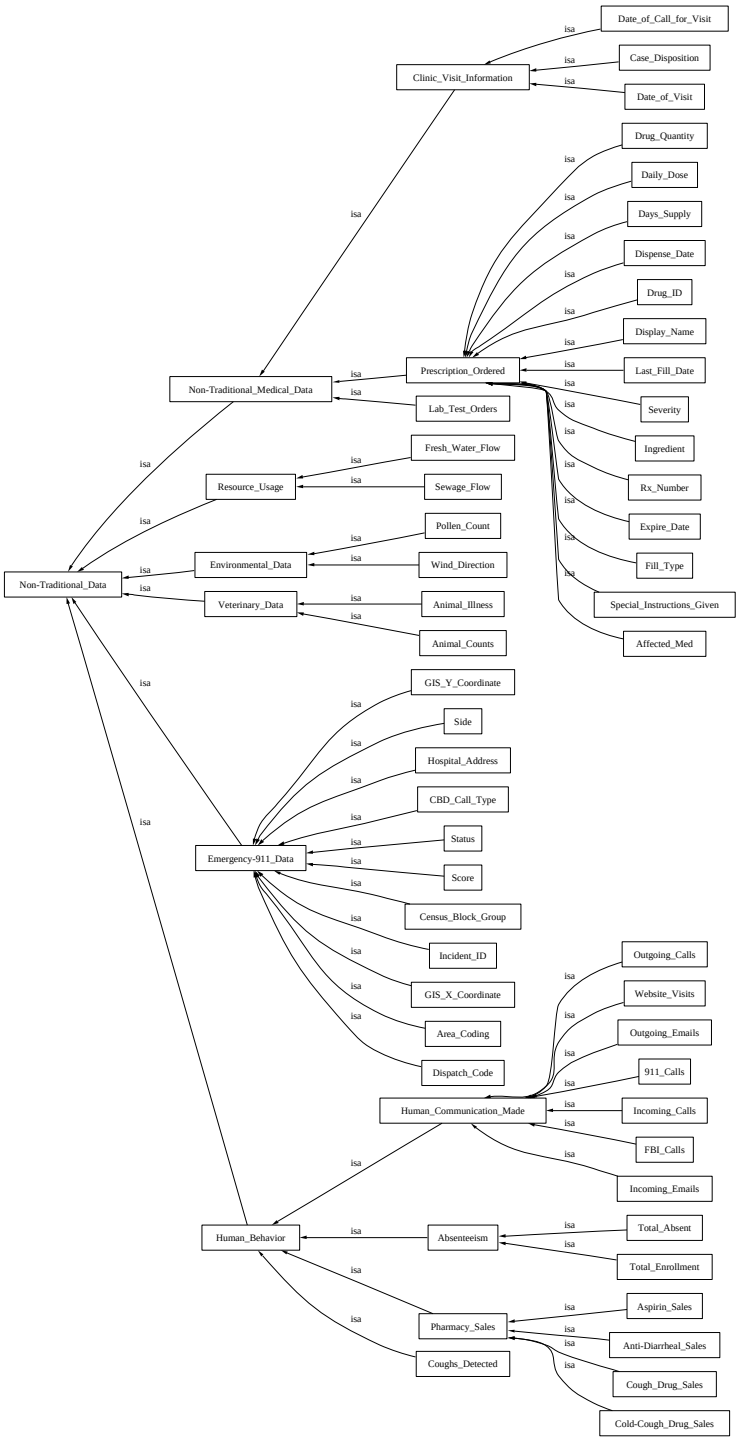


Figure 8.2: A part of the hierarchy of “measured properties” in the BioSAIL ontology.

For determining the number of changes, we use the tool that is described in Section 7.2. We subsequently compare all versions with the succeeding versions and count the number of classes, slots and instances that are added, deleted, renamed or modified. The category of changed frames (i.e. the 'C' columns) *excludes* renamed frames, so a frame that is renamed and otherwise changed (e.g. a slot value changed) is counted twice in the totals. This is also the reason that the number of renamed frames can be higher than the total number of changed frames, e.g. for the classes in the change from 2.0r4 to 2.1r1.

When looking at the table, there are a number of things that attract attention. First, we can see that in general most changes have to do with classes, except for the last revision in which a large number of instances is added. Also, we see that most modifications are additions, followed by changes in the definitions, deletions and renamings. A likely explanation for this is that the ontology is in a development phase, in which extensions are more common than deletions.

Something else that strikes is that there is one revision (from 1.6r4 to 1.6r5) in which nothing has changed. We checked this manually and came to the same conclusion; although the order of definitions is different, the defined classes, slots and instances are identical, and even the size of the files are equal. The time-stamps of the files are more than two weeks apart, so a possible explanation can be that the developer forgot whether he already created a revision and created a new one to be on the safe side.

It is also worth mentioning that most changes typically occur in the first or second revision after a version number change (e.g. from 2.0 to 2.1). A possible explanation for this is the following. The author explained that version number changes were initiated by major changes in the structure. It could be that such structure changes involve other changes, which were either performed in the same revision, or in the revision following the structure change.

As an example for the remainder of this section, we consider the change from version v2.1 release 3 to version 2.1 release 4. We choose this specific transition because it is a change that can relatively easy be understood. The major change is the introduction of another type of "time point". In the first version (i.e. version 2.1r3), there is just one class that models time points, called "Point_in_Time". This concept has slots for day, month, hours, etcetera. In the newer version, an artificial time point is introduced, called "BioWAR_Time_Tick" (to facilitate the recording of time in a simulation). Together with this, a concept "Y-M-D-H-M-S_Point_in_Time" is defined to replace the old concept. Both concept are made a subclass of a new abstract class "Point_In_Time". Figure 8.3 shows the subclasses of "Time_Components" before and after the change.

The change of the time concept has consequences for slots or concepts that use this concept. For example, the range of the slot "Ending" used to be "Point_in_Time", but is "Y-M-D-H-M-S_Point_in_Time" in the new version.

8.1.3 Creating the Change Specification

To produce the change specification, we start with the tool that is described in Section 7.2. This tools does two things: first, it suggests a mapping between the concepts in the old and the new version, and second, based on this mapping, it calculates the required change

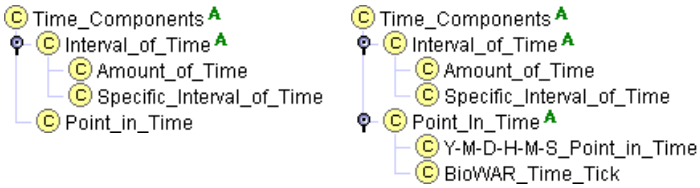


Figure 8.3: The hierarchy of "Time_Components" in the original and the newer version of the ontology.

operations. Some of the calculated change operations are already complex changes. However, as the tool is an extension of the Protégé-tool, its knowledge model is not completely compatible with the knowledge model that we used for our change specification language. To solve this, we manually translate the operations into the ones in our vocabulary. This translation involves the following actions.

- We replace the OKBC operations with the respective OWL operations, e.g., change_own_slot `:VALUE-TYPE` within a slot definition is replaced with Change_Range.
- If a class A has template properties without a facet value, we add A to the domain of these properties. This is because the Protégé knowledge model specifies the applicability of a slot to a class via template slots, whereas in OWL this is done via the "domain" of a slot. For example, the template slot "hour" in a class "Point_in_Time" in the Protégé knowledge model is translated into a property "hour" with "Point_in_Time" as (subset of) the domain.
- We replace the "abstract" and "concrete" roles of classes with an annotation property "role", with an instance "Abstract" and "Concrete" as value, respectively. In this way, we are able to represent an aspect of the Protégé knowledge model that is not part of the OWL knowledge model.
- We ignore numeric minimum and maximum values. We could have represented this via XML Schema datatypes that are specifically defined for a numeric range; however, we decided to leave them out because our set of basic change operations does not include specific operations for changes in numeric minimum and maximum value (as this is not part of the OWL knowledge model). [4]

We translate all operations that are detected by the tool directly into our vocabulary and do not created complex operations ourselves. If the tool detects several changes on one entity, we represent this as a composite change (composed of multiple atomic changes). Appendix C.1 lists the complete change specification between both versions, represented in RDF. Here, we give two examples. The first example is a class addition. This operation is represented in a straightforward way: the operation Add_Class has one property with the complete new definition as its value.

[4] If we would have translated this via XML Schema datatypes, the changes could have been represented as Modify_Range operations from one defined data-type to another. It is even possible to define new complex changes—subclasses of Modify_Range—that specify the change in more detail, e.g. Reduce_Range_of_Datatype_Property etc.

```
<ov:Add_Class>
  <ov:to>
    <owl:Class rdf:about="&new;#Y-M-D-H-M-S_Point_in_Time">
      <rdfs:subClassOf rdf:resource="&new;#Point_In_Time"/>
      <rdfs:comment>
        Instances of this class provide a format to enter a specific
        point in time, at whatever granularity is necessary.
      </rdfs:comment>
    </owl:Class>
  </ov:to>
</ov:Add_Class>
```

Second, we show how a composite change is specified. As described in the meta-ontology in Section 5.5, composite changes are connected to atomic changes via the consists_of property. The example below consists of three atomic changes: a Change_Range_To_Subclass and two Remove_Range operations. The atomic operations specify the old and / or new values of the aspect that is changed via `oldFiller` and `newFiller` properties. Note that in our interpretation of the ontology, the values of the `to` and `from` properties are inherited to the operations where the composite change consists of.

```
<ov:Composite_Change>
  <ov:from rdf:resource="&old;#Expiration_Time"/>
  <ov:to rdf:resource="&new;#Expiration_Time"/>
  <ov:consists_of>
    <ov:Change_Range_To_Subclass>
      <ov:oldFiller rdf:resource="&old;#Point_in_Time"/>
      <ov:newFiller rdf:resource="&new;#Y-M-D-H-M-S_Point_in_Time"/>
    </ov:Change_Range_To_Subclass>
  </ov:consists_of>
  <ov:consists_of>
    <ov:Remove_Range>
      <ov:oldFiller rdf:resource="&old;#Point_in_Time"/>
    </ov:Remove_Range>
  </ov:consists_of>
  <ov:consists_of>
    <ov:Remove_Range>
      <ov:oldFiller rdf:resource="&old;#Amount_of_Time"/>
    </ov:Remove_Range>
  </ov:consists_of>
</ov:Composite_Change>
```

The change specification also contains the “evolution relations” between the entities in the old and the new version of the ontology. For concepts or properties that have changed, their evolution relation is implicitly specified in the operations via the `from` and `to` properties. For the other entities, we use the assumption that an identical fragment identifier in two different files implies an evolution relation (see Section 5.5). For example, if both versions contain a concept that has identifier `#Time_Components`, we assume that there is an evolution relation between the concepts `&old;#Time_Components` and `&new;#Time_Components`.

Note that there are many different valid change specifications for the same change. In the example above, we follow the output of the tool and base our specification on the mappings that it detects. However, as the complete list of changes reveals, it could be that the evolutions that are suggested by the tool are not the ones that actually performed

by the ontology editor. Especially, it is very likely the concept "Point_in_Time" has evolved into "Y-M-D-H-M-S-Point_in_Time", and that a new concept "Point_In_Time" was specified (see again Figure 8.3). This would have resulted in a different change specification that does not have Change_Range_To_Subclass operations for the properties that had "Point_in_Time" as range.[5]

8.1.4 Querying the Change Specification

Now that we have a complete specification of changes represented as instance data of our ontology of change operations, we can exploit the knowledge in that ontology to derive additional information about the changes that have occurred. To illustrate this, we load both the change specification and the ontology of change operations into Sesame (Broekstra et al., 2002a), a system for storing and querying RDF data.

Then, we query the resulting data repository for three things: 1) for the list of entities that have changed, 2) for each entity all operation types that are valid, and 3) the effects of that change—if known. For this, we use the RDF query and transformation language SeRQL (Broekstra and Kampman, 2003b). This language provides constructs to create new RDF statements as a result of a query. We used this feature to retrieve the effect from the ontology of change operations to combine it with the subjects of the change operations.

The complete query and result can be found in Appendix C.2. A part of the output is listed below: it shows that the entity "Beginning" in the old version is changed by the operation Change_Range_To_Subclass, and that the effect (here the effect on classification) is that property becomes more specific. It also shows that the entity "Expiration_Time" is changed by a composite change, which does not give much additional knowledge, and some other changes, like Remove_Range and Change_Range_To_Subclass. Again the effect is that the concept becomes more specific.

```
old:Beginning
    ov:changed_by ov:Change_Range_To_Subclass ;
    ov:has_effect ov:Specialized .

old:Expiration_Time
    ov:changed_by ov:Composite_Change ;
    ov:changed_by ov:Change_Range_To_Subclass ;
    ov:changed_by ov:Composite_Change ;
    ov:changed_by ov:Change ;
    ov:changed_by ov:Composite_Change ;
    ov:changed_by ov:Remove_Range ;
    ov:has_effect ov:Specialized .
```

8.1.5 Summary

In this section, we analyzed an actual change in an ontology evolution process and represented this change within our specification language. As the ontologies used were

[5]This does not mean that the former specification is incrrect, but it can mean that the specification is more verbose and less efficient (e.g., the effects on data validity calculated from this specification might be worse).

available as Protégé files, we were able to use the tool described in the previous chapter to create a change set. This resulted in a list of changes, some of which were implicit or isomorphic. Selecting the changes that were categorized as *directly-changed* and *implicitly-changed* yielded a minimal transformation set in the Protégé terminology. Using a few rules, we were able to translate the Protégé-based operations to the appropriate OWL-based change operations.

The process illustrated how composite changes can be specified by grouping several atomic operations. Also, it showed that basic operations can be intermixed with complex operations, e.g. Change_Range_To_Subclass. It is imaginable how changes can be represented at different levels of granularity by choosing either basic operations or complex operations for specifying the change. Finally, we sketched how the change specification in combination with the ontology of change operations could be used to answer queries about the change.

8.2 Determining Integrity of Ontology Mappings

In Chapter 6, we have described a mechanism to define modular ontologies and mappings between them that allows for local containment of terminological reasoning. This modularization mechanism makes it possible to perform subsumption reasoning within an ontology without having to access other ontologies. This mechanism coincided with a change analysis method that predicts the effect of changes on the concept hierarchy. This method determines whether the changes in one ontology affect the reasoning inside other ontologies or not. Together, these mechanisms allow ontologies to evolve without unpredictable effects on other ontologies.

In this section, we will show how these methods work in a realistic example. For this we use a case study that is undertaken in the WonderWeb project[6].

8.2.1 Ontology in Case Study

In the WonderWeb case study, an existing database schema in the Human Resource (HR) domain is used as the basis for an ontology. The first version of the ontology is created by a tool that automatically converts a schema into an ontology (Volz et al., 2002). In the next phase, the quality of the ontology is improved by relating this ontology to the foundational ontology DOLCE (Gangemi et al., 2002). First, the HR ontology is aligned with the DOLCE ontology, and in several successive steps the resulting ontology is further refined. During this process, the ontology changes continuously, which causes problems when other ontologies refer to definitions in the evolving ontology. Therefore, in our case study, evolution management is important during the entire life-cycle of the ontology development process.

We assume that we have another ontology—we call it the *local ontology*—that uses terms and definitions from the evolving DOLCE+HR ontology—the *external ontology*. As an example, we define a very simple ontology about employees (see Figure 8.4). Our

[6]The WonderWeb project aims at developing scalable infrastructure for the semantic web. For more information, see `http://wonderweb.semanticweb.org/`.

example ontology introduces the concept ‘FulltimeEmployee’ and defines a superclass ‘Employee’ and two subclasses ‘DepartmentMember’ and ‘HeadOfDepartment’ using terms from the DOLCE+HR ontology.

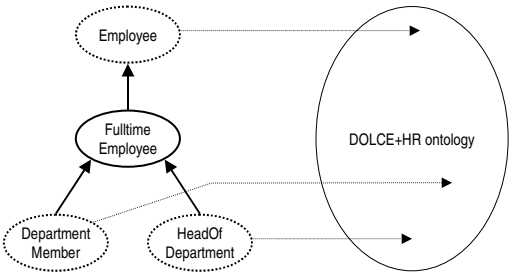


Figure 8.4: A simple ontology (left) with some concepts (dashed ovals) that are defined using terms from the DOLCE+HR ontology (schematically represented by a large oval).

The specific problem in our case is that the changes in the DOLCE+HR ontology could affect the reasoning in the local ontology. We want to be able to predict whether or not the reasoning in the local ontology is still valid for specific changes in the external ontology.

Changes in DOLCE+HR

The evolution of the DOLCE+HR ontology consisted of several steps. Each of these steps involves some typical changes. We will briefly summarize them and show some changes that are typical for a specific step.

In the first step, the extracted HR ontology is aligned with the DOLCE foundational ontology, i.e. the concepts and properties in the HR ontology are connected to concepts and properties in the DOLCE ontology via subsumption relations. For example, the concept ‘Department’ from the HR ontology is made a subclass of ‘Social-Unit’ in DOLCE.

Next, in the refinement step, many changes are made. Some property restrictions are added, and some additional concepts and properties are created to define the HR concepts more precisely. For example, the concept ‘Administrative-Unit’ is introduced as a new subclass of ‘Social-Unit’, and the concept ‘Departments’ is made a subclass of it. Also, the range of the property ‘email’ is restricted from ‘Abstract-Region’ to its new subclass ‘Email’.

In the next step, a number of concepts and properties are renamed to names that better reflect their meaning. For example, ‘Departments’ is renamed to ‘Department’ (singular), and the two different variants of the relation ‘manager_id’ are renamed to ‘employee_manager’ and ‘department_manager’.

In the final step, the tidying step, all properties and concepts that are not necessary anymore are removed and transformed into property restrictions. For example, the property ‘employee_email’ is deleted and replaced by an existential restriction in the class ‘Employee’ on the property ‘abstract_location’ to the class ‘Email’.

8.2.2 Definitions in the Local Ontology

If we consider the local ontology, we have a concept hierarchy that is built up by the following explicitly stated subsumption relations (see Figure 8.4 again):

$$\begin{aligned} FulltimeEmployee &\sqsubseteq Employee \\ DepartmentMember &\sqsubseteq FulltimeEmployee \\ HeadOfDepartment &\sqsubseteq FulltimeEmployee \end{aligned}$$

This ontology introduces 'Full time employee' as a new concept, not present in the case study ontology. Consequently, this concept is only defined in terms of its relation to other concepts in the local ontology.

All other concepts are externally defined in terms of ontology based queries over the case study ontology. The first external definition concerns the concept 'Employee' that is equivalent to the 'Employee' concept in the case study ontology. This can be defined by the following trivial view:

$$Employee \equiv HR : Employee(x)$$

Another concept that is externally defined is the 'Head of Department' concept. We define it to be the set of all instances that are in the range of the 'department manager' relation. The definition of this view given below shows that our approach is flexible enough to define concepts in terms of relations.

$$HeadOfDepartment \equiv HR : \exists y[departmentManager(y, x)]$$

An example for a more complex external concept definition is the concept 'department member' which is defined using a query that consists of three conjuncts, claiming that a department is an employee that is in the has_member relation with a Department.

$$DepartmentMember \equiv HR : \exists y[Department(y) \wedge has\_member(y, x) \wedge Employee(x)]$$

Compiling Implied Knowledge

To allow for local reasoning inside this ontology, we have to derive the subsumption relations between the externally defined concepts, and add these relations to the local ontology.

We immediately see that the definition of Employee subsumes the definition of DepartmentMember, as the former occurs as part of the definition of the latter.

$$\models DepartmentMember \sqsubseteq Employee \tag{8.1}$$

At a first glance, there is no relation between the definition of a Head of Department and the other two statements as it does not use any of the concept or relation names. However, when we use the background knowledge provided by the external ontology we

can derive some implied subsumption relations. The reasoning is as follows. Because the range of the department_manger is set to 'Department' and the domain to 'Manager', the definition of HeadofDepartment is equivalent to:

$$\exists y[Department(y) \wedge department\_manager(y, x) \wedge Manager(x)]$$

As we further know that Manager is a subclass of Employee and department_manager is a sub-relation of has_member, we can derive the following subsumption relation between the externally defined concepts:

$$\models HeadOfDepartment \sqsubseteq Employee \tag{8.2}$$

$$\models HeadOfDepartment \sqsubseteq DepartmentMember \tag{8.3}$$

When the relations 8.1–8.3 are added to the local ontology, it possible to do subsumption reasoning without having to access the DOLCE+HR ontology anymore.

8.2.3 Finding and Characterizing Changes

To find changes in ontologies, we can use one of the tools that is described in the previous chapter. Figure 8.5 shows a screenshot of the OntoView tool when applied to a fragment of the case study ontology. The detected change operations are printed in the left upper corner of each marked change. The figure shows that the old definition of 'Departments' can be transformed into the new definition with three change operations: 1) change the superclass relation from 'Social Unit' to 'Administrative-Unit', 2) change to comment, and 3) add a specific property restriction. Note that the first one is actually a complex operation concept moved down, because 'Administrative-Unit' is a subclass of 'Social Unit'. In the future, the tool should be able to export these changes as RDF instance data for the change ontology.

Now we know the change operations, we can try to give a useful characterization of the effect on the concepts. This is not possible for all concepts. For example, the concept 'Departments', underwent several changes during the whole process: its superclass has changed to a subclass of the original superclass and some property restriction are removed. Both changes have an opposite effect. As a result, we have to characterize the effect of the change as "Unknown". On the contrary, the effect on the relation 'department_manager', is clear: the relation is renamed from 'manager_id'—which has no conceptual effect—and the range is changed form 'Employee' to 'Manager'. Because 'Manager' is a subclass of 'Employee', this change makes it more specific.

Impact on Compiled Relations

The methods described above provide us with abstract information about the impact of changes within an ontology. More specifically, they provide information about the semantic relation between the old and the new version of a concept or a relation.

We now use this procedure to check whether the implied subsumption relations in our case study are still valid. For the sake of simplicity, we restrict us here to relation 8.3:

$$\models HeadOfDepartment \sqsubseteq DepartmentMember$$

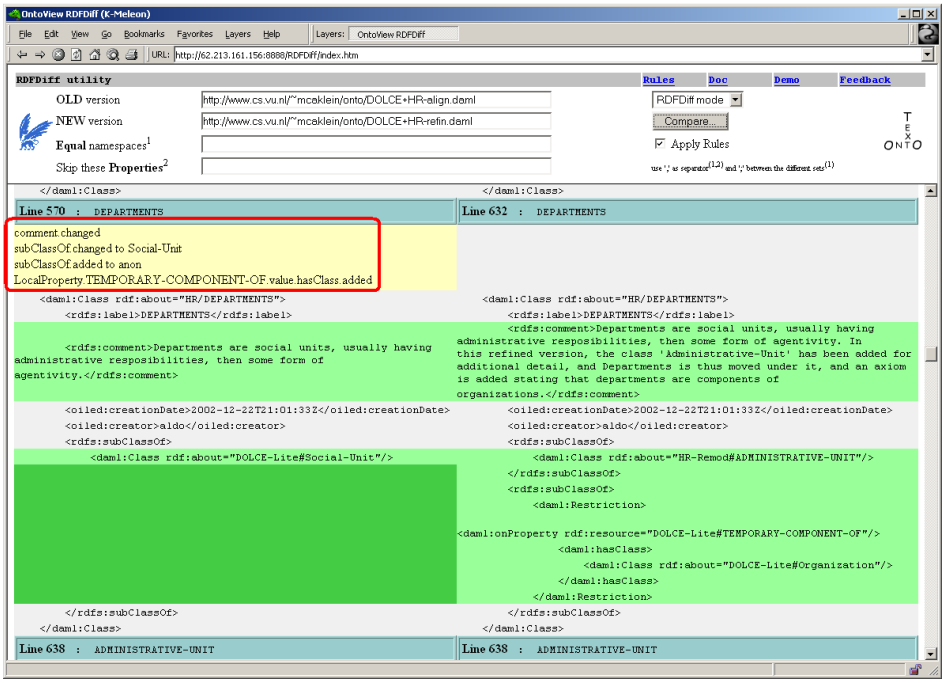


Figure 8.5: Comparison of the aligned and refined version of the DOLCE+HR ontology. The highlighted areas represent changed statements. Note the detected change operations (marked with a box).

For this compiled axiom, the list of 'subsuming' concepts and relations would contain 'Department', 'has_member', and 'Employee', while the list of subsumed concepts and relations would be 'Department', 'department_manager', and 'Manager'. Based on the characterization of these concepts, the procedure concludes that this relation is still valid. We will now illustrate that the conclusions of the procedure are correct by studying the impact of changes mentioned in the problem statement.

Example 1: The Employee Concept The first change we observed is the removal of properties from the Employee concept. Our rules tell that this change makes the new version more general compared to its old version:

$$Employee \sqsubseteq Employee'$$

According to our procedure, this shouldn't be a problem because Employee is in the 'subsuming list'.

When we analyze this change, we see that it has an impact on the definition of the concept DepartmentMember as it enlarges the set of objects allowed to take the first place in the has_member relation. This leads to a new definition of $DepartmentMember'$

with $DepartmentMember \sqsubseteq DepartmentMember'$. As DepartmentMember was already more general than HeadOfDepartment and the Employee concept is not used in the definition of the latter the implied subsumption relation indeed still holds.

Example 2: The department_manager Relation In the second example, we have to deal with a change affecting a relation that is used in an external definition. The relation department_manager is specialized by restricting its range to a more specific concept making it a subrelation of its previous version:

$$department\_manager \sqsupseteq department\_manager'$$

Again, this is harmless according to our procedure, as department_manager is in the 'subsumed list'.

The analysis shows that this change has an impact on the definition of the concept HeadOfDepartment as it restricts the allowed objects to the more specific Class Manager. The new definition $HeadOfDepartment'$ is more specific that the old one: $HeadOfDepartment' \sqsubseteq HeadOfDepartment$. As the old version was already more specific than the definition of DepartmentMember and the department_manager relation is not used in the definition of the latter the implied subsumption is indeed still valid.

Example 3: The Department Concept The different changes of the definition of the department concept left us with no clear idea of the relation between the old and the new version of the concept. In this specific case, however, we can still make assertions about the impact on implied subsumption relations. The reason is that the concept occurs in both the list of subsuming concepts and the list of subsumed concepts. Moreover, it plays the same role in the query concepts, namely restricting the domain of the relation that connects an organizational unit with the set of objects that make up the externally defined concept. As a consequence, the changes have the same impact on both definitions and thus do not invalidate the implied subsumption relation. In general, we can conclude that an implied subsumption relation is still valid if the changed concept occurs in and plays the same role in both definitions involved.

8.2.4 Discussion

In this section, we applied the procedure that was sketched in Section 6.5 in a case study with an ontology that evolved independently. The local ontology that we defined consisted of only a few concepts and relations, in order to keep the example understandable. We designed the local ontology in such a way that we were able to show a number of different situations that could occur. With this carefully designed local ontology, we were able to demonstrate that the procedure can be used to heuristically determine that some changes do not have impact on interfaces that are defined between ontologies. However, it is still a question what the benefits of the procedure are with more complex ontologies. It is clear that more externally defined concepts and more complex local ontologies will result in more "unknown" effects. To compensate for this, we need additional heuristic rules that specify consequences for specific situations, e.g. such as the rule formulated

in example 3 in the previous paragraph. This requires further analysis of the effect of changes in subsumption relations.

8.3 User Study of Change Visualization

In Chapter 7, we presented a tool that implements a visualization mechanism for complex changes. The goal of visualizing changes is to ease the task of analyzing ontology changes. Such an analysis can be useful in several situations, e.g. to validate changes made by somebody else, or to annotate changes with the conceptual relations between versions of constructs. For a visualization of changes, we need both the evolution relation between constructs and *complex* changes. In this section we describe a user study in which we compare the visualization of complex changes with a traditional change visualization.

8.3.1 Aims

Our hypothesis is that this visualization makes it easier for users to analyze the changes that have occurred, compared to traditional visualizations that show the differences concept-by-concept. With this study, we want to provide some evidence for this claim.

As our claim is that *analyzing* changes is easier, we need a task that actually requires users to *understand* the changes. We think that reverse-engineering a previous version of an ontology is such a task. When reverse-engineering a complex change, e.g. the addition of a tree, a user needs to understand that the change is a whole, or he will spend a lot of time in reverse-engineering all the constituting changes one-by-one. If our hypothesis is true, the subjects that use the visualization will spend less time in reverse-engineering the old versions than the subjects that use a traditional side-by-side visualization of the changes.

8.3.2 Methods

We used two groups of subjects for our experiment. We gave both groups a Protégé editor with the *new* version of an ontology loaded. We presented the subjects from one group with the visual-diff, and the subjects in the other group with a text-based representation of the change. We asked both groups to reverse-engineer *old* versions of the ontology, using the given change representation. We measured the time it took them to complete the task and calculated the accuracy afterwards.

As object for the experiment, we used a real ontology evolution scenario, namely a subset of two versions of the UNSPSC ontology. UNSPSC[7] is a standardized hierarchy of products and services that enables users to consistently classify the products and services they buy and sell. We used version 8.0 and 8.4 (in ECCMA numbers[8]) The subset is formed by the classes which have a code that start with a '3'. In version 8.0, this

[7] See `http://www.eccma.org/unspsc/`

[8] The management of UNSPSC has recently been taken over, and the version numbering has changed as well.

selection consist of 3070 classes. The changes between version 8.0 and version 8.4 are the following: 28 class are added (11 of them form a 'subtree addition'), 3 class were removed, 1 subtree is moved, and 8 class are renamed.

For the text-based presentation of the changes, we used a free text diff tool.[9] This tool presents the old and new version side-by-side, and displays the different changes (line added, line changed, line deleted) in different colors.

To prevent that we would compare the wrong thing, i.e. a GUI-based representation of the ontology with a text-based representation, we made some improvements to the text-based representation. First, we translated the ontology in a 'neutral' syntax, to prevent that the obscurity of the RDF syntax would interfere with the results. Each class is defined on two lines: the first line contains the class name and the meta-class between brackets, the second line the superclass of the class. The classes are separated by blank lines. The order of the definitions is a depth-first order of the hierarchy.

```
Resistors   (Class)
  subclass-of Printed_components

Fusistors   (Commodity)
  subclass-of Resistor
```

Second, we reordered the text files in such a way that added classes were actually displayed as added lines, changed classes as changed lines, and deleted classes as deleted lines. If we would have skipped this reordering, the tool would have detected that a block of approximate 2500 lines was added and another block of 2000 lines was deleted (mainly because of a large subtree move). This would have made it unsuitable for a fair comparison.

8.3.3 Subjects

We performed the experiment with 11 subjects. Six of the subjects used the visual-diff for the task, five used a text-diff. The subjects were selected from members of two departments of two different universities, i.e. 7 from the Computer Science department of the *Vrije Universiteit* in Amsterdam (NL) and 4 members from the Medical Informatics department of Stanford University (USA). We selected persons that were familiar with both ontology modeling and the Protégé ontology editor. None of the users had any experience with the visualization itself. Because of technical reasons, all subjects in Stanford performed the task using the visual-diff. Two of the subjects from Amsterdam also used the visual-diff, the others used a text-diff. The difference between both subject groups is less than it seems to be, as both groups consist of international academic workers, with only one Dutch individual in the Amsterdam group and two non-US citizens in the American group.

We gave all users a short (less than 5 minutes) period to make themselves familiar with tools and the specific visualization of the changes. We presented them with three windows: two Protégé editors with an old and a new version of a tiny ontology (10 classes), and one with the appropriate visualization of the changes, i.e. either a text-diff or the visual-diff (see Figure 8.6). This toy example contains several complex changes,

[9] ExamDiff, `http://www.prestosoft.com/ps.asp?page=edp_examdiff`

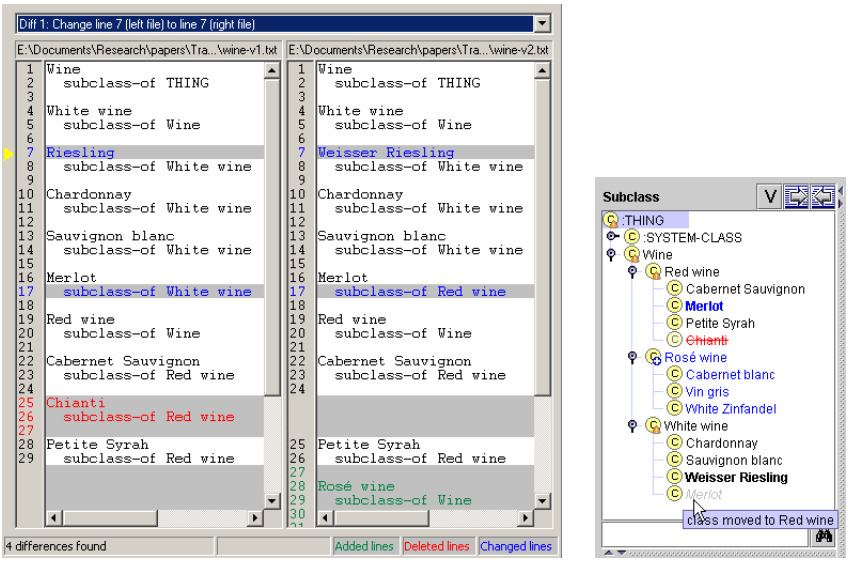


Figure 8.6: The visualizations used for training the subjects. On the left a text-diff of a change in a toy ontology and on the right the visual-diff for the same change.

like trees added, individual classes deleted and added, and classes moved.

We pointed all subjects to the buttons that they could use to navigate through the changes (for both tools), but we did not explain elements of the visualization. All subjects reported within the five minutes that they understood enough of the visualization to start the experiment.

8.3.4 Results and Discussion

In our sample group of 11 users, the average time to complete the whole task was less for the users that used the visual-diff than for the users that used the traditional diff. Table 8.2 gives an overview.

Text-diff	Time	Errors	Visual-diff	Time	Errors
subject 1	20:00	3	subject 6	15:09	1
subject 2	19:52	2	subject 7	10:46	1
subject 3	20:47	2	subject 8	17:22	0
subject 4	20:11	1	subject 9	17:03	0
subject 5	27:54	2	subject 10	20:54	2
			subject 11	25:57	1
Average	*21:45*	*2.0*	*Average*	*17:51*	*0.83*

Table 8.2: The time needed to reverse-engineering the old version of the ontology with the different tools, and the number of errors made.

The average number of errors is more than two times larger for the text-diff users than for the users of visual-diff. This suggests that the visual-diff gives a better *overview* of what actually happened, which reduces the chance to miss changes.

Additionally, the figures show that most time measurements for the text-diff tool are very close to each other, while the *variation* of measurements is much larger for the visual-diff. A possible explanation for this is that the text-diff tool stimulates people to work in a very mechanic manner, in which each change costs a comparable amount of time. The explanation for the larger variation for the visual-diff measurements would then be that different subjects put more or less effort in understanding the changes.

Because the test set is very small, it is difficult to draw reliable conclusions about the significance of the differences found. Nevertheless, we performed unpaired "students *t*-test" to get an impression of the significance of the differences for both the number of errors and the time measurements. The first test tells that the number of errors made with the visual-diff are significantly less than the errors made with the text-diff. The probability of getting the figures in the third and sixth column in Table 8.2 when the two tools are *not* different is only 2.7 %. This means that it is very likely that the tools have a different effect on the number of errors made. Another test for the time measurements tells that the difference is not significant. If the test is performed for all measurements, we get a probability of 18.7% for *wrongly* concluding that the tools are different. Even when we omitted the outliers, i.e. the measurements that were more than 2 times the standard deviation different from the average, we find that this probability is still 7.6%. This is too high to safely conclude that the visual-diff is better.

Altogether, we can say that the figures suggest that the visual-diff helps users in understanding the change, but the experiment is too small to draw strong conclusions. A more comprehensive analysis, with larger data groups or with other experiments, for example based on think-aloud protocols, is necessary.

Chapter 9

Conclusions

In this final chapter of the thesis, we look both back and forward. We look back by summarizing the key points of the thesis (Section 9.1) and by reviewing the research questions with which we started in the first chapter (Section 9.2). In Section 9.3, we look forward and discuss open issues and future research directions.

9.1 Key Points and Conclusions

The thesis starts with a description of the "Semantic Web", a foreseen extension of the current World Wide Web that contains explicit knowledge descriptions (i.e. ontologies) which can by used by computers. As this extension will have the same decentralized and uncontrolled organization as the current generation of the web, we will have a situation in which not only the information on the web changes continuously, but also the knowledge that is used to interpret it.

To understand the artifacts and relations that play a role in this situation, we then discuss different languages that are meant for publishing data and knowledge on the web (Chapter 2). The datamodels that underly the languages dictate their respective role in the Semantic Web. XML functions as an general encoding mechanism for both data and knowledge, and XML Schema can be used to prescribe its structure. RDF is a basic datamodel that is used as a means to represent semantic relationships between "things" on the web, called *resources*. RDF Schema is used to specify categories of things and additional knowledge about relationships. We show how RDF Schema can be extended to use more expressive knowledge representation languages for describing RDF relationships and data.

We analyze the problems caused by evolving ontologies in three different ways (see Chapter 3). First, we perform a literature study and provide an overview of more than ten aspects in which ontologies can differ. We distinguish differences between ontology languages from differences between the ontologies themselves. The latter can be further divided into differences in the things that are described and differences in the *way in which* things are described. This distinction appears to be important for determining

what effect changes have. Second, we compare ontology evolution with database schema versioning. This reveals that both are in theory not very different: for example, schemas for object-oriented databases can be seen as ontologies, and distributed databases are not centrally maintained. However, the facts that i) *in practice* ontologies have richer data models, ii) that they are by nature developed in a decentralized way, and iii) that they are often used as data themselves, justify a specific approach for ontology change management. Such an approach has to deal with a larger set of possible changes, would have to account for different usage scenarios, and allow for incomplete information about the changes. As a third technique to analyze the problems caused by evolving ontologies, we have conducted a series of interviews with maintainers of large ontologies. These interviews give us some insight in the methods that are applied in practice. In appears that commonly used techniques are: i) dividing ontologies into separately maintained parts, ii) minimizing changes or restricting the types of changes that are allowed, and iii) using databases for low-level versioning support.

All this results in three major assumptions on which we built a framework for further exploring change management methods for distributed ontologies. The first assumption is that there are different levels at which an ontology can be interpreted (i.e. the conceptualization, specification, and representation level). It can happen that changes are performed or detected at one level, but that the consequences are determined by changes in another level. As there is not necessarily a one-to-one relation between changes at different levels, sometimes human knowledge about a change is required to decide about the influence on other interpretation levels. Second, the effect of changes varies for different tasks in which ontologies are used. Web-site navigation, accessing data, and subsumption reasoning are differently affected by changes in an ontology. The third assumption is that there are many different, and possibly incomplete ways in which changes can be specified. We can not presuppose that one of these is present. For example, we may have just two distinct ontology specifications. Based on these assumptions, we have designed a change process model that describes at an abstract level how to handle ontology change. Also, we have developed a representation for ontology change that can function as intermediary language. The main element of this representation is a so-called "transformation set", which is a set of operations that completely specify the change.

As a vocabulary for the change representation, we propose a taxonomy of change operations (see Chapter 5). Because it is influenced by the expressivity of the ontology language considered, the set of operations is to some extent language specific. We derived the set by iterating over all the elements in the *meta-model* of the ontology language, creating "add", "delete" and—when appropriate—"modify" operations for all elements. In this way, we abstracted from representational issues and had a guarantee that we covered all possible modifications. To decide on which language we would base our change representation, we compare two well-known knowledge representation formalisms: the OKBC knowledge model and the OWL (Full) ontology language. By comparing their respective knowledge models, we conclude that strictly speaking neither of these is a subset of the other. However, it appears that the things that are not present in OWL are quite rare in practice, for example *local* equivalence or inverse restrictions on properties. Therefore, we decide to use OWL as basis for our change operations.

In addition to the operations that are directly derived from the knowledge model of the

ontology language, we also introduce *complex operations*. These operations can be used to group together several basic operations, and/or to encode additional characteristics of the change operations. Operations that cluster other operations can be used when the constructing operations form a logical unit (e.g. removing something and adding it somewhere else), and when the composite effect of operations is different from the effect of operations on their own. Operations that encode additional knowledge can be used to define specialized variants of other operations, e.g. an operation that specifies that the range of a property is *restricted* instead of just *modified*. Complex operations are useful for both visualizing and understanding changes and for determining their effect. The possibility to define complex changes forms an extension mechanism that allows for task- or domain-specific representations of change.

The framework consists—besides a representation for changes—also of an abstract process model for ontology change management (Chapter 6). Basically, this model describes the following steps: 1) change information should be created from the sources that are available, 2) heuristics, algorithms or human input should be used to enrich this information (e.g. resulting in a set of change operations), and 3) ontology evolution related tasks can be performed with help of the enriched change information. Together with others we developed two tools that can be used to create change information (step 1). We also specify several processes for deriving new information from existing change information (step 2). In addition, we describe how to perform four ontology evolution related tasks (from step 3). First, we explain how we can use an ontology to access or interpret instance data of another version of the ontology. Second, we describe a procedure that heuristically determines the validity of mappings between ontology modules. This procedure predicts whether subsumption reasoning within one module is still valid if changes have occurred in an ontology from which concepts or relations are imported. Third, we adapt a methodology for the synchronization of related, but independently evolving ontologies to be used within our framework. Finally, we show a tool that visualizes changes at an abstract level to help people with understanding these.

In the final part of the thesis, we describe three practical studies. These studies do not evaluate the framework as a whole, but they apply particular elements in a realistic setting. First, we analyze an actual evolution history of an ontology that describes medical information sources. This study gives us insight in the distribution of different types of changes (additions, deletions, modification) in the different phases of its development. We use the developed change representation to specify one particular transition in the evolution history. The study also sketches how the ontology of change operations, when annotated with effects of operations on specific tasks, can be used to query for the effect of a particular change. Second, we apply the procedure for determining the integrity of mappings between ontology modules on a small but realistic ontology evolution. For this, we use an ontology that is extracted from a database schema and refined in several successive steps. This study reveals that the heuristics predict the effects correctly, but it also questions the usability of the procedure for more complex mappings and ontologies. Finally, we perform a user study to evaluate a high-level visualization of changes. We have visualized changes in a large (around 3000 classes) subset of the UNSPSC ontology and gave users a tasks that required them to understand what had changed. The study suggests that the visualization can indeed help with understanding changes. The users

performed the task with significant less errors and—on average—in less time

9.2 Reviewing the Research Questions

In the introductory chapter, we formulated the following central research question.

> "Which mechanisms and methods are required to cope with ontology change in a dynamic and distributed setting, where ontologies are used as means to improve computerized information exchange?"

The overview of key issues above already provided a partial answer to this question. Hereafter, we will discuss the three smaller questions in which the general question was split up.

What are the specific characteristics of change management for distributed ontologies?

When answering this question, it is useful to distinguish between characteristics that are related to the nature of ontologies themselves, and characteristics related to the uncontrolled and distributed setting in which they are typically used and evolve. The following characteristics are related to the nature of ontologies:

- Change-management procedures for ontologies should take into account that there are different levels of interpretation for ontologies. This has as a consequence that the appearance of the change (i.e. the change in representation or specification) can not be used as only source for determining the consequences. For each change one should consider whether it propagates to other levels of interpretation, i.e. whether it is also a change in the specification or in the conceptualization.
- There is not one specific characteristic of an ontology that a change management procedure should try to preserve, because ontologies can be used for different tasks. A general procedure that prescribes required follow-up changes to maintain a valid state can therefore not be developed. Instead, the consequences of changes for specific ontology use-cases should be considered.
- The expressivity and the semantics of specific ontology languages influence the details of the change-management procedures. The fact that ontology languages are typically quite expressive gives many different types of changes. The formal semantics behind an ontology language can sometimes be exploited to solve specific problems, for example in determining the logical relation between old and new versions of concepts.

Characteristics that are related to the distributed usage of ontologies are:

- A change-management method for ontologies can not assume that specific information about a change is present or can be acquired. Sometimes the change history might even be unknown. The change management methods should thus be able to work with incomplete and sometimes even inaccurate information.

- It is not known who uses which ontology version for which purpose. This means that ontology change management procedures should not only consider the relation between the most recent and the "previous" version, but take the relations between all versions into account.

What is an adequate representation of changes between ontologies?

The function of a change specification is to exchange information about change between users, tools, or individual processes in the change management task. In Chapter 4, we described several representations of ontology change. Each of these encodes knowledge about the change that is potentially useful in particular situations. Based on this, we can conclude that a comprehensive change specification consists of at least the following pieces of information about an ontology change:

- an operational specification of change, i.e. the required steps to transform the old version into the new version;
- the conceptual relation between old and new versions of constructs in an ontology;
- meta-data about the change;
- the evolution relation between constructs in the old and new version;
- task or domain specific consequences of changes.

For the operational specification of the change (the transformation), sets of *change operations* can be used. This is a common way to specify changes in database versioning as well. Ontology change operations specify modifications to an ontology in a precise and unambiguous way. Because a specification via operations can be verbose, we propose a *minimal transformation set* as an efficient specification for transformations. This is a set of ontology-change operations that applied to the old version of the ontology results in the new version, in which all operations are required to achieve the new version.

The operations that are required to specify all possible changes can be derived from the meta-model of the ontology language that is used. By iteration over all elements of the meta-model, we can guarantee that every change can be specified. We discovered that it can be beneficial to use also specialized operations and composite operations. This makes it possible to specify a different characteristic for a specific variant of the operation, or for the combination of the constituting operations. We propose a mechanism that can be used to extend the set of defined basic operations with additional and specific *complex operations*.

It appeared that a transformation set on its own can be used as basis for deriving several of the other aspects of a change specification. The minimal transformation is therefore an efficient and concise representation for changes, although not complete (because not all information about a change can be derived from it).

What methods and techniques can be developed to solve possible problems caused by ontology change?

Because of the characteristics described above, it is not possible to design a general procedure for ontology change management. Instead, there is a need for a variety of

methods that start from different types of change information and that support particular tasks. In this thesis we described a framework in which several of such methods have a place. There are three different roles for methods or techniques: i) generating change information, ii) deriving additional information about the change, and iii) solving specific problems in specific situations.

We found the following methods and techniques to be useful.

Comparison algorithms Comparison algorithms are used to find the specific changes that occured between two ontology versions.

Ontology mapping When there is no trace of changes between ontology versions and there are also no persistent identifiers for concepts, mapping techniques are useful to find out which concept evolved into which other concept.

Reasoning services Subsumption reasoning is helpful to determine conceptual relations between versions of concepts, whereas consistency checks can be used to evaluate the state of a changed ontology.

Human validation In some situations input from humans experts is required to explicate the reasons behind a change, for example to specify whether the conceptualization has changed.

Change visualization A visualization of changes at a higher level of abstraction can help users or developers to understand changes, especially when one "abstract change" involves multiple smaller changes.

Effect Prediction Heuristics The ultimate effect of changes is sometimes difficult to determine because the effects of multiple changes can interfere, and because the process can be computationally expensive. Heuristics that look at basic change operations are sometimes useful to predict or exclude effects.

Guidelines For some change related tasks, for example data translation or ontology synchronization, it is possible to formulate guidelines or protocols that prescribe steps to be taken to prevent problems or to achieve a specific result.

Besides these, new methods and techniques can be added to the framework if specific tasks ask for it, or if new types of change information become available.

9.3 Outlook

One of the things that become clear throughout this thesis is that traditional database management solutions are not sufficient for change management of distributed ontologies. Instead, specific techniques and methods are needed. In this thesis, we explore the problem of ontology change and sketch a general framework that can be used to relate ontology-evolution tasks and methods. As part of this, we outlined several methods that solve some envisioned problems and perform particular tasks.

The outlined procedures still need to be fine-tuned and evaluated for concrete tasks. Many of the described methods seem to work better for relatively small changes than for larger changes. Therefore, the evaluations of the procedures should also consider the practical usability: is the amount of changes between ontology versions in practice small enough to derive usable results?

The developed framework describes a limited number of tasks. It can and should be extended with other evolution tasks and other types of change information. Analyzes of specific ontology-evolution use-cases are required to come up with additional and specific tasks related to ontology change. New ontology management tools, for example tools that support collaborative development, can also be reasons for extending the framework.

A basic assumption in this thesis is that ontologies evolve in a distributed and uncontrolled setting. In such a development scenario, the social issues around ontology change management are by definition not relevant because all developers work independent of each other. Therefore, we do not discuss procedures for collaborative development. However, in practice we see that many large ontologies are still maintained in a (somewhat) centralized way. In these situations, social procedures are relevant. Further research is needed to find answers to questions related to best practices for proposing changes, validating changes suggested by others, reaching mutual agreement, and resolving conflicts when different changes are proposed by different parties.

The assumption about a distributed and uncontrolled setting is motivated by the idea of the Semantic Web. From one perspective, this makes the future applicability of our work slightly uncertain. The idea of the Semantic Web still has to shape up, so we can not yet tell what the exact role of ontologies will be, and how such ontologies will be developed. We will have to wait for the answers to these questions before we can tell whether change management as we sketched it in this thesis is an important issue, or that other problems than the ones we envisaged will be important.

However, when looking at this from a wider perspective, we see that the research in this thesis does not rely on the Semantic Web. Independent from the unfolding of the Semantic Web, we observe a trend in computer science towards distributed computing, which means that several devices at different locations are involved in performing computerized tasks. The internet itself, where many computers co-operate to transport and process data, is probably the largest example of this. It becomes even more apparent with the idea of web-services, which are basically tasks that are executed on other computers via the Web. We can also see this trend at a smaller scale, for example with the rise of intelligent personal devices, like mobile phones, PDA's, digital camera's, MP3-players, and even "smart clothing". More and more, these devices are able to connect to each other, whether wireless or wired, and co-operate or exchange information. Yet another example of this trend is called grid-computing, which is the concept that computing power can be requested where needed and provided where available. What is common in all these situations is that computerized devices co-operate and exchange information with previously unknown other devices, while each of them uses information in a particular way. This can be partly facilitated by standardization of communication protocols and data formats, but will increasingly require semantic descriptions of data and tasks. This makes another trend in computer science: semantic descriptions are getting more important. Also in

database research we discover developments towards semi-structured formats based on XML and distributed usage of databases over the web, which require distributed semantic descriptions of the contents. Given the inevitable evolution of applications and data, together with the trends towards distributed computing and the increasing role of semantic descriptions, we expect that change management for distributed knowledge structures will become even more important in the future.

Appendix A

Guideline for Interviews

Ontology itself

1. What is the goal of the ontology?
2. How many people are using the ontology?
3. Is there instance data attached to the ontology?
    (a) If yes, is the instance data centralized or distributed?
    (b) Do you know all the places were instance data of the ontology resides?
4. Is the ontology modularized?
    (a) What is the criterion for modularization?

Ontology development

5. When did the development of the ontology start?
6. What is your role with respect to the development of this ontology?
7. Who else is involved in the development of this ontology?
    (a) What is their role?
8. Who provides the content knowledge of the ontology?
9. Who does the modeling?

Current change management

10. How often do changes occur?
11. Who is performing the changes?
12. Are changes validated by somebody?
    (a) By who?
13. Are changes verified?
    (a) How?
    (b) By who?
14. If yes to 4, has the modularization criterion ever changed?
    (a) How often?
    (b) From what to what?
    (c) Why?
15. Do you apply some versioning system, for example “assigning versioning numbers”, “using a source control system”?
    (a) Can you explain how it works? For example, when do you check in, or how do you assign version numbers?
16. Are there official releases?
    (a) Is there a predefined schedule for the releases?
    (b) How often are there releases?
    (c) How are official releases published?

Reasons for change

17. Could you mention specific motives for performing changes?
18. I’ve looked at several of your ontologies and discovered several types of change.
    (a) Are there changes in the ontology that do not fall in one of those categories?
    (b) Can you describe them?
    (c) Which types occur most often? And second, third often?
19. Are different types of change handled in different ways? For example, I can imagine that modeling changes force a major update, while bugs only give a minor release.

Problems caused by changes

20. Did the evolution of the ontology cause problems for you?

    (a) If yes, what kind of problems?

    (b) How did you solve them?

    (c) Were there problems that you couldn't solve?

21. Did you take any measures in advance to prevent possible problems that could come with the evolution? (E.g. prefixes for names to assure uniqueness, etc.)

22. If "distributed" answered to 3.a, were the changes / new releases announced to the users?

    (a) Do you know whether they updated their instance data?

    (b) If yes, were they forced to update?

Desired support form versioning systems

23. What kind of support from a versioning system would be helpful for you?

24. I listed several kinds of support. Could you please explain which are would be helpful for and which won't?

Closing

25. Finally, are there some things that you think might be relevant but are not covered in this interview?

Attachments

Prompt question 18

Different types of change:

- changing the modeling choices (e.g. introducing meta-classes)
- restructuring domain knowledge
- extensions to the domain
- correcting errors

Prompt question 24

Aspects of versioning support:

- Data accessibility
  being able to access older instance data via a newer version of the ontology (or vice versa)
- Consistent reasoning
  being able to get know whether specific types of reasoning provide the same answer over the new ontology as they did over the old
- Synchronization
  being able to update local copies of ontologies according to changes that occurred in a remote one
- Data translation
  being able to update instance data so that it conforms with a new version of an ontology
- Management of development
  being able to verify and authorize changes step by step
- Editing support
  being able to edit changes in the ontology without losing specific values, facets or instance data
- Propagation support
  being able to automatically delete or update frames in the ontology if other parts are changed

Appendix B

Ontology of Change Operations

The table below lists all classes that represent basic change operations. The complete ontology, including all complex operations, can be found at `http://ontoview.org/changes/2/1`. The last column in the table tells for which variant of OWL the operations are relevant: F stands for OWL Full, D for OWL DL, and L for OWL Lite.

Object	Operation	Argument(s)	Variant
Ontology	Add_Class	Class definition	FDL
Ontology	Remove_Class	Class ID	FDL
Class	Add_Superclass	Class ID	FDL
Class	Remove_Superclass	Class ID	FDL
Class	Modify_Superclass	Two class ID's	FDL
Class	Add_Equivalent_Class	Class description or ID	FDL
Class	Remove_Equivalent_Class	Class description or ID	FDL
Class	Modify_Equivalent_Class	Two class descriptions or ID's	FDL
Class	Add_Disjoint_Class	Class description or ID	FD
Class	Remove_Disjoint_Class	Class description or ID	FD
Class	Modify_Disjoint_Classs	Two class descriptions or ID's	FD
Class	Add_Restriction	Restriction & S or SF	FDL
Class	Remove_Restriction	Restriction	FDL
ValueRestriction	Change_To_Universal_Restriction	–	FDL
ValueRestriction	Change_To_Existential_Restriction	–	FDL
ValueRestriction	Modify_Restriction_Filler	Class description or ID's	FDL
CardinalityRestriction	Add_Lowerbound	Integer	FDL
CardinalityRestriction	Remove_Lowerbound	Integer	FDL
CardinalityRestriction	Modify_Lowerbound	Two integers	FDL
CardinalityRestriction	Add_Upperbound	Integer	FDL
CardinalityRestriction	Remove_Upperbound	Integer	FDL
CardinalityRestriction	Modify_Upperbound	Two integers	FDL
Resource	Add_Type	Class ID	F
Resource	Remove_Type	Class ID	F
Resource	Modify_Type	Two class ID's	F
Resource	Add_Label	Value	FDL
Resource	Remove_Label	Value	FDL
Resource	Modify_Label	Two values	FDL

continued on next page

continued from previous page

Object	**Operation**	**Argument(s)**	**Variant**
Resource	Add_Comment	Value	FDL
Resource	Remove_Comment	Value	FDL
Resource	Modify_Comment	Two values	FDL
Resource	Add_Annotation	Property ID & Value	FDL
Resource	Remove_Annotation	Property ID & Value	FDL
Resource	Modify_Annotation	Property ID & Two values	FDL
Individual	Add_Equivalent_Individual	Individual	FDL
Individual	Remove_Equivalent_Individual	Individual	FDL
Individual	Modify_Equivalent_Individual	Two individuals	FDL
Individual	Add_Disjoint_Individual	Individual	FDL
Individual	Remove_Disjoint_Individual	Individual	FDL
Individual	Modify_Disjoint_Individual	Two individuals	FDL
Ontology	Add_Individual	Individual definition	FDL
Ontology	Remove_Individual	Individual ID	FDL
Ontology	Add_Property	Property definition	FDL
Ontology	Remove_Property	Property ID	FDL
Property	Add_Domain	Class description or ID	FDL
Property	Remove_Domain	Class description or ID	FDL
Property	Modify_Domain	Class description or ID	FDL
Property	Add_Range	Class description or ID	FDL
Property	Remove_Range	Class description or ID	FDL
Property	Modify_Range	Class description or ID	FDL
Property	Set_Functionality	–	FDL
Property	Unset_Functionality	–	FDL
Property	Add_Symmetry	–	FDL
Property	Remove_Symmetry	–	FDL
Property	Set_Transitivity	–	FDL
Property	Unset_Transitivity	–	FDL
Property	Set_InverseFunctionality	–	FDL
Property	Unset_InverseFunctionality	–	FDL
Property	Add_Superproperty	Property ID	FDL
Property	Remove_Superproperty	Property ID	FDL
Property	Modify_Superproperty	Two property ID's	FDL
Property	Add_Equivalent_Property	Property ID	FDL
Property	Remove_Equivalent_Property	Property ID	FDL
Property	Modify_Equivalent_Property	Two property ID's	FDL
Property	Add_Inverse_Property	Property ID	FDL
Property	Remove_Inverse_Property	Property ID	FDL
Property	Modify_Inverse_Property	Two property ID's	FDL
Property	Change_To_DatatypeProperty	Property ID	FDL
Property	Change_To_ObjectProperty	Property ID	FDL

Appendix C

Change Specification for BioSAIL ontology

C.1 Change Specification between v2.1r3 and v2.1r4

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
     <!ENTITY old  "http://www-smi.stanford.edu/projects/biostorm/v21r3" >
     <!ENTITY new  "http://www-smi.stanford.edu/projects/biostorm/v21r4" >
   ]>

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:owl="http://www.w3.org/2002/07/owl#"
         xmlns:ov="http://ontoview.org/changes/2/1#">

<ov:Add_Class>
  <ov:to>
    <owl:Class rdf:about="&new;#Compass_Direction">
      <rdfs:subClassOf rdf:resource="&new;#Kind-of-Property"/>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:onProperty rdf:resource="&new;#Unit"/>
          <owl:allValuesFrom rdf:resource="&new;#Compass_Direction_Units"/>
        </owl:Restriction>
      </rdfs:subClassOf>
    </owl:Class>
  </ov:to>
</ov:Add_Class>

<ov:Add_Class>
  <ov:to>
    <owl:Class rdf:about="&new;#Compass_Direction_Units">
      <rdfs:subClassOf rdf:resource="&new;#Allowed_Units"/>
    </owl:Class>
  </ov:to>
```

```
</ov:Add_Class>

<ov:Add_Class>
  <ov:to>
    <owl:Class rdf:about="&new;#Environmental_Measurement">
      <rdf:type rdf:resource="&new;#Data_Provider_Metaclass"/>
      <rdfs:subClassOf rdf:resource="&new;#Data_Provider"/>
    </owl:Class>
  </ov:to>
</ov:Add_Class>

<ov:Add_Class>
  <ov:to>
    <owl:Class rdf:about="&new;#Y-M-D-H-M-S_Point_in_Time">
      <rdfs:subClassOf rdf:resource="&new;#Point_In_Time"/>
      <rdfs:label>
        Instances of this class provide a format to enter a specific
        point in time, at whatever granularity is necessary.
      </rdfs:label>
    </owl:Class>
  </ov:to>
</ov:Add_Class>

<ov:Remove_Label>
  <ov:from rdf:resource="&old;#Point_in_Time"/>
  <ov:to rdf:resource="&new;#Point_In_Time"/>
  <ov:oldFiller>
    Instances of this class provide a format to enter a specific point
    in time, at whatever granularity is necessary.
  </ov:oldFiller>
</ov:Remove_Label>

<ov:Change>
  <ov:from rdf:resource="&old;#Weather_Data"/>
  <ov:to rdf:resource="&new;#Wind_Direction"/>
</ov:Change>

<ov:Change_Range_To_Subclass>
  <ov:from rdf:resource="&old;#Beginning"/>
  <ov:to rdf:resource="&new;#Beginning"/>
  <ov:oldFiller rdf:resource="&old;#Point_in_Time"/>
  <ov:newFiller rdf:resource="&new;#Y-M-D-H-M-S_Point_in_Time"/>
</ov:Change_Range_To_Subclass>

<ov:Change_Restriction_Filler_To_Subclass>
  <ov:from rdf:resource="&old;#DateTime_Datum"/>
  <ov:to rdf:resource="&new;#DateTime_Datum"/>
  <ov:oldFiller>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&old;#Datum_Contents"/>
      <owl:allValuesFrom rdf:resource="&old;#Point_in_Time"/>
    </owl:Restriction>
  </ov:oldFiller>
  <ov:newFiller>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&new;#Datum_Contents"/>
```

```
      <owl:allValuesFrom rdf:resource="&new;#Y-M-D-H-M-S_Point_in_Time"/>
    </owl:Restriction>
  </ov:newFiller>
</ov:Change_Restriction_Filler_To_Subclass>

<ov:Change_Range_To_Subclass>
  <ov:from rdf:resource="&old;#Ending"/>
  <ov:to rdf:resource="&new;#Ending"/>
  <ov:oldFiller rdf:resource="&old;#Point_in_Time"/>
  <ov:newFiller rdf:resource="&new;#Y-M-D-H-M-S_Point_in_Time"/>
</ov:Change_Range_To_Subclass>

<ov:Composite_Change>
  <ov:from rdf:resource="&old;#Expiration_Time"/>
  <ov:to rdf:resource="&new;#Expiration_Time"/>
  <ov:consists_of>
    <ov:Change_Range_To_Subclass>
      <ov:oldFiller rdf:resource="&old;#Point_in_Time"/>
      <ov:newFiller rdf:resource="&new;#Y-M-D-H-M-S_Point_in_Time"/>
    </ov:Change_Range_To_Subclass>
  </ov:consists_of>
  <ov:consists_of>
    <ov:Remove_Range>
      <ov:oldFiller rdf:resource="&old;#Point_in_Time"/>
    </ov:Remove_Range>
  </ov:consists_of>
  <ov:consists_of>
    <ov:Remove_Range>
      <ov:oldFiller rdf:resource="&old;#Amount_of_Time"/>
    </ov:Remove_Range>
  </ov:consists_of>
</ov:Composite_Change>

<ov:Composite_Change>
  <ov:from rdf:resource="&old;#Measurement"/>
  <ov:to rdf:resource="&new;#Measurement"/>
  <ov:consists_of>
    <ov:Remove_Property_Restriction>
      <ov:oldFiller>
        <owl:Restriction>
          <owl:onProperty rdf:resource="&old;#Expiration_Time"/>
          <owl:allValuesFrom rdf:resource="&old;#Point_in_Time"/>
        </owl:Restriction>
      </ov:oldFiller>
    </ov:Remove_Property_Restriction>
  </ov:consists_of>
  <ov:consists_of>
    <ov:Remove_Property_Restriction>
      <ov:oldFiller>
        <owl:Restriction>
          <owl:onProperty rdf:resource="&old;#Expiration_Time"/>
          <owl:allValuesFrom rdf:resource="&old;#Amount_of_Time"/>
        </owl:Restriction>
      </ov:oldFiller>
    </ov:Remove_Property_Restriction>
  </ov:consists_of>
```

```
  <ov:consists_of>
    <ov:Add_Property_Restriction>
      <ov:newFiller>
        <owl:Restriction>
          <owl:onProperty rdf:resource="&new;#Expiration_Time"/>
          <owl:allValuesFrom rdf:resource="&new;#Time_Components"/>
        </owl:Restriction>
      </ov:newFiller>
    </ov:Add_Property_Restriction>
  </ov:consists_of>
</ov:Composite_Change>

<ov:Composite_Change>
  <ov:from rdf:resource="&old;#Specific_Interval_of_Time"/>
  <ov:to rdf:resource="&new;#Specific_Interval_of_Time"/>
  <ov:consists_of>
    <ov:Change_Restriction_Filler_To_Subclass>
      <ov:oldFiller>
        <owl:Restriction>
          <owl:onProperty rdf:resource="&old;#Beginning"/>
          <owl:allValuesFrom rdf:resource="&old;#Point_in_Time"/>
        </owl:Restriction>
      </ov:oldFiller>
      <ov:newFiller>
        <owl:Restriction>
          <owl:onProperty rdf:resource="&new;#Beginning"/>
          <owl:allValuesFrom rdf:resource="&new;#Y-M-D-H-M-S_Point_in_Time"/>
        </owl:Restriction>
      </ov:newFiller>
    </ov:Change_Restriction_Filler_To_Subclass>
  </ov:consists_of>
  <ov:consists_of>
    <ov:Change_Restriction_Filler_To_Subclass>
      <ov:oldFiller>
        <owl:Restriction>
          <owl:onProperty rdf:resource="&old;#Ending"/>
          <owl:allValuesFrom rdf:resource="&old;#Point_in_Time"/>
        </owl:Restriction>
      </ov:oldFiller>
      <ov:newFiller>
        <owl:Restriction>
          <owl:onProperty rdf:resource="&new;#Ending"/>
          <owl:allValuesFrom rdf:resource="&new;#Y-M-D-H-M-S_Point_in_Time"/>
        </owl:Restriction>
      </ov:newFiller>
    </ov:Change_Restriction_Filler_To_Subclass>
  </ov:consists_of>
</ov:Composite_Change>

<ov:Change_Range_To_Subclass>
  <ov:from rdf:resource="&old;#Valid_Until_Time"/>
  <ov:to rdf:resource="&new;#Valid_Until_Time"/>
  <ov:oldFiller rdf:resource="&old;#Point_in_Time"/>
  <ov:newFiller rdf:resource="&new;#Y-M-D-H-M-S_Point_in_Time"/>
</ov:Change_Range_To_Subclass>
```

```
</rdf:RDF>
```

C.2 Querying Changes and Effects

C.2.1 Query

```
construct DISTINCT Object <ov:changed_by> OperationType;
                          <ov:changed_by> NestedOperationType;
                          <ov:has_effect> Effect
    from Operation <ov:from> Object;
                   <serql:directType> OperationType
                                [<ov:effect> Effect],
        [Operation <ov:consists_of> NestedOperation
                                <serql:directType> NestedOperationType
                                [<ov:effect> Effect]]

using namespace
   rdf = <!http://www.w3.org/1999/02/22-rdf-syntax-ns#>,
   rdfs = <!http://www.w3.org/2000/01/rdf-schema#>,
   owl = <!http://www.w3.org/2002/07/owl#>,
   ov = <!http://ontoview.org/changes/2/1#>,
   old = <!http://www-smi.stanford.edu/projects/biostorm/v21r3#>,
   new = <!http://www-smi.stanford.edu/projects/biostorm/v21r4#>
```

C.2.2 Result

The formatting of the result is slightly edited for the purpose of readability. The RDF data is represented in the N3-syntax (Berners-Lee, 1998).[1]

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix new: <http://www-smi.stanford.edu/projects/biostorm/v21r4#> .
@prefix old: <http://www-smi.stanford.edu/projects/biostorm/v21r3#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ov: <http://ontoview.org/changes/2/1#> .

old:Point_in_Time
    ov:changed_by ov:Change ;
    ov:changed_by ov:Remove_Label .

old:Weather_Data
    ov:changed_by ov:Change .

old:Beginning
    ov:changed_by ov:Change_Range_To_Subclass ;
    ov:has_effect ov:Specialized .

old:DateTime_Datum
    ov:changed_by ov:Change ;
    ov:changed_by ov:Change_Restriction_Filler_To_Subclass .

old:Ending
```

[1] For an easy to read explanation, see `http://www.w3.org/2000/10/swap/Primer`.

```
    ov:changed_by ov:Change_Range_To_Subclass ;
    ov:has_effect ov:Specialized .

old:Expiration_Time
    ov:changed_by ov:Composite_Change ;
    ov:changed_by ov:Change_Range_To_Subclass ;
    ov:changed_by ov:Composite_Change ;
    ov:changed_by ov:Change ;
    ov:changed_by ov:Composite_Change ;
    ov:changed_by ov:Remove_Range ;
    ov:has_effect ov:Specialized .

old:Measurement
    ov:changed_by ov:Composite_Change ;
    ov:changed_by ov:Change ;
    ov:changed_by ov:Composite_Change ;
    ov:changed_by ov:Remove_Property_Restriction ;
    ov:changed_by ov:Composite_Change ;
    ov:changed_by ov:Add_Property_Restriction .

old:Specific_Interval_of_Time
    ov:changed_by ov:Composite_Change ;
    ov:changed_by ov:Change ;
    ov:changed_by ov:Composite_Change ;
    ov:changed_by ov:Change_Restriction_Filler_To_Subclass .

old:Valid_Until_Time
    ov:changed_by ov:Change_Range_To_Subclass ;
    ov:has_effect ov:Specialized .
```

Samenvatting

Het *internet* is een geweldig succesvolle toepassing van computertechnologie. Waar het World Wide Web tien jaar geleden nog voornamelijk een hulpmiddel voor academici was, is het in 2004 als informatiebron niet meer weg te denken uit de maatschappij. Dit succes verhult echter dat de rol van computers bij het vergaren van informatie via het internet heel beperkt is. Computers zorgen voor de opslag, het transport en het weergeven van gegevens, maar aan het selecteren, combineren of interpreteren van informatie dragen ze nauwelijks bij. Dit veroorzaakt bijvoorbeeld de vaak gehoorde klacht "dat je veel te veel terugkrijgt bij het zoeken". De reden dat computers weinig behulpzaam zijn bij het selecteren en combineren van informatie is tweeledig. Ten eerste is de informatie op webpagina's vrijwel altijd geschreven voor mensen; voor computers is de informatie niet meer dan een rij niet-verwerkbare letters. Ten tweede vereisen zulke taken vaak achtergrondkennis over het onderwerp van de informatie, en computers hebben deze kennis van zichzelf niet.

Sinds enkele jaren wordt daarom op wereldwijde schaal onderzoek gedaan naar een "nieuwe generatie" internet: een uitbreiding van het huidige internet waarbij achtergrondkennis is toegevoegd en de informatie zo gestructureerd is dat computers het kunnen verwerken. Deze nieuwe versie van het internet wordt het *semantische web* genoemd. Naast pagina's met informatie voor mensen en computers, bevat het semantische web ook documenten die achtergrondkennis over een bepaald onderwerp specificeren. Deze specificaties worden in de informatica *ontologieën* genoemd, afgeleid van het begrip met dezelfde naam in de filosofie. De in de informatica gebruikte ontologieën hebben vaak de vorm van een hiërarchische indeling van categorieën en subcategorieën en hun eigenschappen. Een eenvoudige ontologie zou bijvoorbeeld kunnen beschrijven dat een "Student" een subcategorie van "Persoon" is, met als specifieke eigenschap dat er een relatie "ingeschreven_aan" met een "Universiteit" is.

Dit proefschrift gaat over het omgaan met wijzigingen in op het internet gepubliceerde ontologieën. Veranderingen in de kennisbeschrijvingen kunnen namelijk invloed hebben op de geldigheid van de bewerkingen die computers met de informatie uitvoeren. Het beheersen van de effecten van veranderingen in kennisspecificaties is geen nieuw probleem, ook bij het onderhouden van kennis-gebaseerde computerprogramma's en databanken komt dit voor. De specifieke moeilijkheid in het semantisch web is dat het internet een "anarchistische" structuur heeft, in de zin dat iedereen in staat is te publiceren wat hij wilt en dat er geen centrale autoriteit is die procedures kan afdwingen. Veel klassieke oplossingen voor het omgaan met veranderingen zijn gebaseerd op het

volgen van bepaalde wijzigingsprocedures en zijn daarom niet toepasbaar op het internet.

Om dit probleem aan te pakken bestuderen we in dit proefschrift allereerst de verschillende computertalen die worden gebruikt voor het publiceren van gegevens en kennis op het internet. Daarbij kijken we vooral naar de datamodellen die aan de verschillende talen ten grondslag liggen, omdat deze bepalen wat feitelijk vastgelegd wordt wanneer te taal gebruikt wordt om gegevens te beschrijven. De *eXtensible Markup Language* (XML) is een taal die als algemeen coderingsmechanisme voor zowel gegevens als kennis gebruikt kan worden, waarbij alleen deel–onderdeel relaties worden vastgelegd. *XML Schema* is een taal waarmee de indeling van XML documenten beschreven kan worden. De RDF taal (*Resource Description Framework*) maakt het mogelijk om de eigenschappen te beschrijven van objecten die een adres op het internet hebben, zoals webpagina's, maar ook boeken, of zelfs begrippen die genoemd worden in kennisbeschrijvingen. Met behulp van deze taal kan bijvoorbeeld gespecificeerd worden dat een document een bepaalde auteur heeft, of dat de relatie "is getrouwd met" op mensen van toepassing is. De eenvoud van RDF is tegelijkertijd de kracht van de taal: doordat eigenschappen van zowel concrete objecten als van abstracte begrippen—waaronder elementen van de taal zelf—kunnen worden beschreven, kan de taal gebruikt worden om een complexere taal te definiëren. Dat is reeds gedaan met *RDF Schema*, een taal waarmee categorieën van objecten kunnen worden gedefinieerd en extra informatie over relaties kan worden beschreven. Wij laten zien hoe een computertaal die kennis nog gedetailleerder kan beschreven kan worden gedefinieerd met behulp van RDF als uitbreiding van RDF Schema.

Na het bestuderen van de talen, analyseren we als tweede stap de problemen die door wijzigingen in ontologieën worden veroorzaakt. We doen dit op drie verschillende manieren. Ten eerste beschrijven we op basis van een literatuurstudie meer dan tien verschillende aspecten waarin ontologieën van elkaar kunnen verschillen, en dus waarin ze kunnen veranderen. Het blijkt onder andere relevant te zijn om onderscheid te maken tussen een verandering in de kennis zelf en een verandering in de manier waarop de kennis is beschreven. Ten tweede vergelijken we het veranderingsproces van schema's voor databanken met het veranderingsproces van ontologieën. Het blijkt dat beide theoretisch niet veel van elkaar verschillen, maar dat sommige problemen bij het beheren van databanken in de praktijk nauwelijks een rol spelen, terwijl die problemen bij het beheren van ontologieën vrijwel altijd aanwezig zijn. Wij beargumenteren dat daarom een aparte methodiek voor het omgaan met wijzigingen in ontologieën noodzakelijk is. Zo'n methodiek moet rekening houden met de verschillende taken waarvoor ontologieën gebruikt kunnen worden en kunnen werken met incomplete informatie over wijzigingen. Als derde onderdeel van de probleemanalyse hebben we een aantal interviews afgenomen bij beheerders van grote ontologieën. Hieruit hebben we een aantal gangbare technieken voor het omgaan met wijzigingen kunnen destilleren, zoals het opdelen van ontologieën in delen die apart worden beheerd en het verbieden van bepaalde soorten wijzigingen.

Vanuit de analyse van de problemen formuleren we drie uitgangspunten voor een methodiek voor het omgaan met wijzigingen in ontologieën op het internet. Het eerste uitgangspunt is dat ontologieën op meerdere interpretatie niveaus beschouwd kunnen worden, namelijk het conceptualisatie-niveau, het specificatie-niveau en het represen-

tatie-niveau. Wijzigingen die in één van de niveaus worden aangebracht of gedetecteerd, kunnen gevolgen hebben op één van de andere niveaus. Omdat er geen één-op-één relatie is tussen veranderingen op de verschillende niveaus, is soms een menselijk oordeel nodig om de consequenties van veranderingen te bepalen. Het tweede uitgangspunt is dat het effect van wijzingen verschilt voor de verschillende taken waarvoor een kennisbeschrijving wordt gebruikt. Redeneertaken kunnen op een andere manier worden beïnvloed door wijzigingen dan het opvragen van informatie. Het derde uitgangspunt is dat er veel verschillende manieren zijn om informatie over wijzigingen te specificeren, maar dat we er op in het semantisch web niet vanuit kunnen gaan dat een van deze representaties beschikbaar is.

Op basis van deze uitgangspunten ontwikkelen we methodiek die bestaat uit een procesmodel voor het omgaan met veranderingen in ontologieën en een taal waarmee wijzigingen beschreven kunnen worden. Het centrale onderdeel van een wijzigingsbeschrijving is een "transformatie-set", een verzameling van wijzigingsoperaties die de verandering volledig beschrijft.

Welke wijzigingen op een ontologie mogelijk zijn hangt af van de ontologietaal die wordt gebruikt. De verzameling van alle mogelijke wijzigingsoperaties creëren we door naar het kennismodel van de ontologietaal te kijken. Zo'n model definieert welke kenniselementen (zoals categorieën, relaties, eigenschappen, eigenschappen van relaties) met een taal beschreven kunnen worden. Door voor ieder kenniselement "*add*", "*remove*" en "modify" operaties te definiëren, verkrijgen we een verzameling operaties waarmee alle mogelijke wijzigingen beschreven kunnen worden. We werken deze operaties uit voor de ontologietaal OWL (*Web Ontology Language*), omdat uit een vergelijking met een andere veel gebruikt kennismodel OKBC (*Open Knowledge Base Connectivity*) blijkt dat het kennismodel van OKBC praktisch bevat is in dat van OWL.

Als aanvulling op de wijzigingsoperaties die afgeleid zijn van het kennismodel van de ontologietaal—de basale operaties—introduceren we zogenaamde "complexe operaties". Dit zijn operaties die een samenvoeging zijn van verschillende basale operaties of die extra informatie over de wijziging bij zich dragen. Het kan nuttig zijn om basale operaties te groeperen wanneer de wijzigingen een logische eenheid vormen (bijvoorbeeld het verwijderen van iets op de ene plek en het weer toevoegen op een andere plek), of wanneer het effect van de operaties tezamen anders is dan het gecombineerde effect van de afzonderlijke operaties. Operaties die extra kennis over een wijzigingen bij zich dragen kunnen worden gebruikt om specifieke varianten van basale operaties te definiëren, bijvoorbeeld een operatie die niet alleen specificeert dat het bereik van een bepaalde eigenschap veranderd is, maar ook dat het *beperkt* is. Deze complexe operaties zijn onder andere nuttig bij het visualiseren van veranderingen. De mogelijkheid om specifieke complexe operaties te definiëren is een mechanisme dat het mogelijk maakt om domein- of taakspecifieke veranderingsspecificaties te creëren.

Naast een taal voor het specificeren van wijzigingen, bevat de methodiek ook een procesmodel voor het omgaan met veranderingen in ontologieën. Dit model beschrijft de volgende stappen: 1) de beschikbare informatie over de wijzigingen wordt verzameld, 2) heuristieken, afleidingsregels en inschattingen van mensen worden gebruikt om deze informatie aan te vullen, 3) met behulp van deze informatie worden veranderingsgevoelige taken uitgevoerd. In het kader van dit onderzoek zijn twee computer-

programma's ontwikkeld die helpen bij het verzamelen van informatie over wijzigingen (stap 1). Voor het uitvoeren van stap 2 beschrijven we verschillende recepten voor het aanvullen van de informatie. Daarnaast laten we zien hoe vier veranderingsgevoelige taken kunnen worden uitgevoerd met behulp van de gegenereerde wijzigingsinformatie (stap 3). We leggen allereerst uit hoe gegevens die beschreven zijn via een oudere versie van een ontologie via een nieuwere versie beschikbaar gemaakt kunnen worden. Ten tweede beschrijven we hoe zonder veel rekentijd bepaald kan worden of vertalingen tussen ontologieën nog geldig zijn nadat één van de twee gewijzigd is. Als derde laten we zien hoe een bestaande methode voor het "in de pas" laten lopen van twee versies van ontologieën die onafhankelijk van elkaar worden beheerd, kan worden gebruikt binnen onze methodiek. De vierde taak die gebruik maakt van de veranderingsinformatie is het visualiseren van wijzigingen. We beschrijven een computerprogramma dat wijzigingen op een abstract niveau kan tonen aan gebruikers.

In het laatste deel van het proefschrift beschrijven we drie praktische studies. Deze studies evalueren niet de methodiek als geheel, maar laten zien hoe sommige elementen van de methodiek in een realistische setting kunnen worden toegepast. In de eerste studie beschouwen we een bestaande ontwikkelingsgeschiedenis van een ontologie over medische informatiebronnen. Hiermee krijgen we een indruk van de verdeling van soorten wijzigingen (toevoegingen, verwijderingen) in de verschillenden fases van de ontwikkeling van een ontologie. We laten zien hoe één specifieke versie-verandering kan worden beschreven met behulp van de ontwikkelde taal. In de tweede praktische studie passen we de methode om voor het bepalen van de geldigheid van een vertaling tussen ontologieën te bepalen toe op een klein maar realistisch veranderingsproces in een ontologie. Ten slotte beschrijven we een gebruikersevaluatie van een abstracte visualisatie van wijzigingen in een grote ontologie (ongeveer 3000 categorieën). De uitkomst suggereert dat de visualisatie inderdaad helpt bij het overzien van de veranderingen.

De komende jaren moet blijken of en hoe het semantisch web vorm zal krijgen. Hoewel het werk dat in dit proefschrift beschreven wordt zijn motivatie vindt in het semantisch web, hangt de relevantie niet af van het al of niet slagen van het semantisch web. Onmiskenbaar is in de informatica een trend waarneembaar naar systemen waarbij verschillende computers met elkaar samenwerken. Hierbij valt te denken aan *web-services* (kleine taken die door andere computers op het internet worden uitgevoerd), maar ook aan de vele elektronische apparaatjes die met elkaar kunnen communiceren, zoals mobieltjes, digitale camera's en MP3-spelers. Om met elkaar te kunnen samenwerken, moeten deze computers gegevens uitwisselen met voorheen "onbekende" systemen. Dit kan deels worden mogelijk gemaakt door standaardisatie van communicatieprotocollen, maar zal ook vragen om expliciete semantische beschrijvingen van de data. Gegeven de onvermijdelijke veranderingen waarin zulke systemen onderhevig zijn, zal het omgaan met wijzigingen in kennisbeschrijvingen een steeds belangrijker onderwerp worden.

Bibliography

Ashburner, M., Ball, C. A., Blake, J. A., Botstein, D., Butler, H., Cherry, J. M., Davis, A. P., Dolinski, K., Dwight, S. S., Eppig, J. T., Harris, M. A., Hill, D. P., Issel-Tarver, L., Kasarskis, A., Lewis, S., Matese, J. C., Richardson, J. E., Ringwald, M., Rubin, G. M., and Sherlock, G. (2000). Gene Ontology: tool for the unification of biology. *Nature Genetics*, 25:25–29.

Baader, F., Calvanese, D., McGuiness, D., Nardi, D., and Patel-Schneider, P., editors (2003). *The Description Logic Handbook - Theory, Implementation and Applications*. Cambridge University Press.

Banerjee, J., Kim, W., Kim, H.-J., and Korth, H. F. (1987). Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *SIGMOD Record (Proc. Conf. on Management of Data)*, 16(3):311–322.

Batini, C., Lenzerini, M., and Navathe, S. B. (1986). A comparative analysis of methodologies of database schema integration. *ACM Computing Surveys*, 18(4):323–364.

Bechhofer, S., Goble, C., and Horrocks, I. (2001). DAML+OIL is not enough. In *Proceedings of the International Semantic Web Working Symposium (SWWS)*, Stanford University, California, USA.

Bechhofer, S., Horrocks, I., Patel-Schneider, P. F., and Tessaris, S. (1999). A proposal for a description logic interface. In Lambrix, P., Borgida, A., Lenzerini, M., Möller, R., and Patel-Schneider, P., editors, *Proceedings of the International Workshop on Description Logics (DL'99)*, pages 33–36, Linköping, Sweden.

Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., and Stein, L. A. (2004). OWL web ontology language reference. W3c recommendation, World Wide Web Concortium.

Beckett, D. (2003). RDF/XML syntax specification (revised). W3c working draft, World Wide Web Concortium.

Berliner, B. (1990). CVS II: Parallelizing software development. In USENIX Association, editor, *Proceedings of the Winter 1990 USENIX Conference*, pages 341–352, Washington, DC, USA. USENIX.

Berners-Lee, T., Fielding, R., and Masinter, L. (1998). RFC 2396: Uniform Resource Identifiers (URI): Generic syntax. Status: DRAFT STANDARD.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):34–43.

Berners-Lee, T. (1998). Notation 3 – ideas about web architecture. `http://www.w3.org/DesignIssues/Notation3.html`. Design Issues.

Bernstein, P. A., Halevy, A. Y., and Pottinger, R. A. (2000). A vision for management of complex models. *SIGMOD Record*, 29(4):55–63.

Borgida, A. and Serafini, L. (2002). Distributed description logics: Directed domain correspondences in federated information sources. In Meersman, R. and Tari, Z., editors, *On The Move to Meaningful Internet Systems 2002: CoopIS, Doa, and ODBase*, volume 2519 of *LNCS*, pages 36–53. Springer Verlag.

Borst, P. (1997). *Construction of Engineering Ontologies for Knowledge Sharing and Reuse*. PhD thesis, Universiteit Twente.

Bowers, S. and Delcambre, L. (2000). Representing and transforming model-based information. In *First Workshop on the Semantic Web at the Fourth European Conference on Digital Libraries*, Lisbon, Portugal.

Brickley, D. and Guha, R. V. (2000). Resource Description Framework (RDF) Schema Specification 1.0. Candidate recommendation, World Wide Web Consortium.

Brickley, D., Hunter, J., and Lagoze, C. (1999). ABC: A logical model for metadata interoperability. Harmony discussion note, see `http://www.ilrt.bris.ac.uk/discovery/harmony/docs/abc/abc_draft.html`.

Broekstra, J., Kampman, A., and van Harmelen, F. (2002a). Sesame: An architecture for storing and querying RDF and RDF Schema. In Horrocks, I. and Hendler, J. A., editors, *Proceedings of the First International Semantic Web Conference (ISWC 2002)*, volume 2342 of *Lecture Notes in Computer Science*, pages 54–68, Sardinia, Italy. Springer-Verlag.

Broekstra, J. and Kampman, A. (2003a). Inferencing and truth maintenance in RDF Schema: exploring a naive practical approach. In *Workshop on Practical and Scalable Semantic Systems (PSSS), ISWC 2003*, Sanibel Island, Florida, USA.

Broekstra, J. and Kampman, A. (2003b). SeRQL user manual. Technical report, Aidministrator BV.

Broekstra, J., Klein, M., Decker, S., Fensel, D., van Harmelen, F., and Horrocks, I. (2001). Enabling knowledge representation on the web by extending RDF schema. In *Proceedings of the 10th World Wide Web conference*, pages 467–478, Hong Kong, China. ACM Press.

Broekstra, J., Klein, M., Decker, S., Fensel, D., van Harmelen, F., and Horrocks, I. (2002b). Enabling knowledge representation on the Web by extending RDF Schema. *Computer Networks*, 39(5):609–634.

Buckeridge, D. L., Graham, J., O'Connor, M. J., Choy, M. K., Tu, S. W., and Musen, M. A. (2002). Knowledge-based bioterrorism surveillance. In *American Medical Informatics Association Symposium*, San Antonio, TX, USA.

Cadoli, M. and Donini, F. M. (1997). A survey on knowledge compilation. *AI Communications*, 10(3-4):137–150.

Chalupsky, H. (2000). OntoMorph: A translation system for symbolic logic. In Cohn, A. G., Giunchiglia, F., and Selman, B., editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 471–482, San Francisco, CA. Morgan Kaufmann.

Chaudhri, V. K., Farquhar, A., Fikes, R., Karp, P. D., and Rice, J. P. (1998a). OKBC: A programmatic foundation for knowledge base interoperability. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 600–607, Menlo Park. AAAI Press.

Chaudhri, V. K., Farquhar, A., Fikes, R., Karp, P. D., and Rice, J. (1998b). Open knowledge base connectivity 2.0. Technical report, Knowledge Systems Laboratory Stanford University.

Cimino, J. J. (1996). Formal descriptions and adaptive mechanisms for changes in controlled medical vocabularies. *Methods of Information in Medicine*, 35(3):202–210.

Connolly, D., van Harmelen, F., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., and Stein, L. A. (2001). DAML+OIL (March 2001) Reference Description. W3C Note, World Wide Web Consortium.

Donini, F. M., Lenzerini, M., Nardi, D., and Schaerf, A. (1996). Reasoning in description logics. In Brewka, G., editor, *Principles of Knowledge Representation*, Studies in Logic, Language and Information, pages 193–238. CSLI Publications.

Elsevier/Embase, editor (2003). *EMTREE 2003: The Life Science Thesaurus*. Elsevier Science Ltd, paperback edition.

Farquhar, A., Fikes, R., and Rice, J. (1997). The ontolingua server: A tool for collaborative ontology construction. *International Journal of Human-Computer Studies*, 46(6):707–727.

Fensel, D., Horrocks, I., van Harmelen, F., Decker, S., Erdmann, M., and Klein, M. (2000). OIL in a nutshell. In Dieng, R. and Corby, O., editors, *Knowledge Engineering and Knowledge Management; Methods, Models and Tools, Proceedings of the 12th International Conference EKAW 2000*, number 1937 in LNCS, pages 1–16, Juan-les-Pins, France. Springer-Verlag.

Fensel, D. and Musen, M. A. (2001). The semantic web: A new brain for humanity. *IEEE Intelligent Systems*, 16(2).

Fergerson, R. W., Noy, N. F., and Musen, M. A. (2000). The knowledge model of Protégé-2000: Combining interoperability and flexibility. In Dieng, R. and Corby, O., editors, *Knowledge Engineering and Knowledge Management; Methods, Models and Tools, Proceedings of the 12th International Conference EKAW 2000*, number 1937 in Lecture Notes in Artificial Intelligence, Juan-les-Pins, France. Springer-Verlag.

Franconi, E., Grandi, F., and Mandreoli, F. (2000). A semantic approach for schema evolution and versioning in object-oriented databases. In *Computational Logic 2000*, number 1861 in Lecture Notes in Computer Science, pages 1048–1062.

Gangemi, A., Guarino, N., Masolo, C., Oltramari, A., and Schneider, L. (2002). Sweetening ontologies with DOLCE. In *13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02)*, volume 2473 of *Lecture Notes in Computer Science*, page 166 ff, Sigüenza, Spain.

Goldstein, M. K., Coleman, R. W., Tu, S. W., Shankar, R. D., O'Connor, M. J., Musen, M. A., Martins, S. B., Lavori, P. W., Shlipak, M. G., Oddone, E., Advani, A. A., Gholami, P., and Hoffman, B. B. (2004). Translating research into practice: Sociotechnical integration of automated decision support for hypertension in three medical centers. *Journal of American Medical Informatics Association*. E-publication ahead of print.

Grosso, W. E., Gennari, J. H., Fergerson, R. W., and Musen, M. A. (1998). When knowledge models collide (how it happens and what to do). In *Proceedings of the 11th Workshop on Knowledge Acquisition, Modeling and Management (KAW '98)*, Banff, Canada.

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2).

Haarslev, V. and Moller, R. (2001). Description of the RACER system and its applications. In *Proceedings of the Description Logics Workshop DL-2001*, pages 132–142, Stanford, CA.

Halevy, A. Y. (2001). Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294.

Heflin, J. and Hendler, J. (2000). Dynamic ontologies on the web. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, pages 443–449. AAAI/MIT Press, Menlo Park, CA.

Hendler, J. and McGuinness, D. L. (2000). The DARPA agent markup language. *IEEE Intelligent Systems*, 16(6):67–73.

Horrocks, I., Fensel, D., Broekstra, J., Decker, S., Erdmann, M., Goble, C., van Harmelen, F., Klein, M., Staab, S., Studer, R., and Motta, E. (2000). OIL: The Ontology Inference Layer. Technical Report IR-479, Vrije Universiteit Amsterdam, Faculty of Sciences. See http://www.ontoknowledge.org/oil/.

Horrocks, I. and Tessaris, S. (2000). A conjunctive query language for description logic aboxes. In *AAAI/IAAI*, pages 399–404.

Katz, R. H. (1990). Towards a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408.

Kitakami, H., Mori, Y., and Arikawa, M. (1996). An intelligent system for integrating autonomous nomenclature databases in semantic heterogeneity. In *Database and Expert System Applications, DEXA'96*, number 1134 in Lecture Notes in Computer Science, pages 187–196, Zürich, Switzerland.

Klahold, P., Schlageter, G., and Wilkes, W. (1986). A general model for version management in databases. In Chu, W. W., Gardarin, G., Ohsuga, S., and Kambayashi, Y., editors, *Twelfth International Conference on Very Large Data Bases*, pages 319–327, Kyoto, Japan. Morgan Kaufmann.

Klein, M., Fensel, D., Kiryakov, A., and Ognyanov, D. (2002a). Ontology versioning and change detection on the web. In *13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02)*, number 2473 in LNCS, page 197 ff, Sigüenza, Spain.

Klein, M., Fensel, D., van Harmelen, F., and Horrocks, I. (2001a). The Relation between Ontologies and XML Schemas. *Linköping Electronic Articles in Computer and Information Science*, 6(4).

Klein, M., Kiryakov, A., Ognyanov, D., and Fensel, D. (2002b). Finding and characterizing changes in ontologies. In *Proceedings of the 21st International Conference on Conceptual Modeling (ER2002)*, number 2503 in LNCS, pages 79–89, Tampere, Finland.

Klein, M. and Noy, N. F. (2003). A component-based framework for ontology evolution. In *Proceedings of the Workshop on Ontologies and Distributed Systemsm, IJCAI '03*, Acapulco, Mexico. Also available as Technical Report IR-504, Vrije Universiteit Amsterdam.

Klein, M. and Stuckenschmidt, H. (2003). Evolution management for interconnected ontologies. In *Workshop on Semantic Integration at ISWC 2003*, Sanibel Island, Florida.

Klein, M. (2001a). Combining and relating ontologies: an analysis of problems and solutions. In Gomez-Perez, A., Gruninger, M., Stuckenschmidt, H., and Uschold, M., editors, *Workshop on Ontologies and Information Sharing, IJCAI'01*, Seattle, USA.

Klein, M. (2001b). XML, RDF, and Relatives (short tutorial). *IEEE Intelligent Systems, special issue on "Semantic Web Technology"*, 16(2):26–28.

Klein, M. (2003). *Knowledge Annotation for the Semantic Web*, chapter Interpreting XML via an RDF Schema. IOS Press, Amsterdam.

Klein, T. E., Chang, J. T., Cho, M. K., Easton, K. L., Fergerson, R., Hewett, M., Lin, Z., Liu, Y., Liu, S., Oliver, D. E., Rubin, D. L., Shafa, F., Stuart, J. M., and Altman, R. B. (2001b). Integrating genotype and phenotype information: An overview of the PharmGKB project. *The Pharmacogenomics Journal*, 1:167–170.

Lassila, O. and Swick, R. R. (1999). Resource Description Framework (RDF): Model and Syntax Specification. Recommendation, World Wide Web Consortium. See http://www.w3.org/TR/REC-rdf-syntax/.

Lerner, B. S. (2000). A model for compound type changes encountered in schema evolution. *ACM Transactions on Database Systems*, 25(1):83–127.

Maedche, A., Motik, B., and Stojanovic, L. (2003). Managing multiple and distributed ontologies on the semantic web. *The VLDB Journal*, 12:286–302.

Marco, D. (2000). *Building and Managing the Meta Data Repository: A Full Lifecycle Guide*. Wiley & Sons.

McGuinness, D. L. and van Harmelen, F. (2004). OWL web ontology language overview. W3c recommendation, World Wide Web Concortium.

McGuinness, D. L. (2000). Conceptual modelling for distributed ontology environment. In *Proceedings of the Eighth International Conference on Conceptual Structures Logical, Liguistic, and Computational Issues (ICCS2000)*, Darmstadt, Germany.

McIlraith, S. and Amir, E. (2001). Theorem proving with structured theories. In Nebel, B., editor, *Proceedings of the International Joint Conference on Artificial Intelligence IJCAI'01*, pages 624–634, San Mateo. Morgan Kaufmann.

Noy, N. F. and Klein, M. (2003). Visualizing changes during ontology evolution. In *Collected Posters ISWC 2003*, Sanibal Island, Florida, USA.

Noy, N. F. and Klein, M. (2004). Ontology evolution: Not the same as schema evolution. *Knowledge and Information Systems*, 6(4):428–440.

Noy, N. F., Kunnatur, S., Klein, M., and Musen, M. A. (2004). Tracking changes during ontology evolution. In *3rd International Semantic Web Conference (ISWC2004)*, Hiroshima, Japan.

Noy, N. and Musen, M. (2002). PROMPTDIFF: A fixed-point algorithm for comparing ontology versions. In *18th National Conference on Artificial Intelligence (AAAI-2002)*, Edmonton, Canada.

Ognyanov, D. and Kiryakov, A. (2002). Tracking changes in RDF(S) repositories. In *13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02)*, number 2473 in LNCS, page 373ff, Sigüenza, Spain.

Oliver, D. E., Shahar, Y., Musen, M. A., and Shortliffe, E. H. (1999). Representation of change in controlled medical terminologies. *Artificial Intelligence in Medicine*, 15(1):53–76.

Oliver, D. E. (2000). *Change Management and Synchronization of Local and Shared Versions of a Controlled Vocabulary*. PhD thesis, Stanford University.

Pinto, H. S. and Martins, J. P. (2002). Evolving ontologies in distributed and dynamic settings. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, Toulouse, France.

Proper, H. A. and van der Weide, T. P. (2000). A general theory for the evolution of application models. *IEEE Transactions on Knowledge and Data Engineering*, 7(6).

Rahm, E. and Bernstein, P. A. (2001). A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4).

Rector, A. (2003). Modularisation of domain ontlogies implemented in description logics and related formalisms including OWL. In *Proceedings of the 16th International FLAIRS Conference*. AAAI.

Robson, C. (2001). *Real World Research*. Blackwell Publishers, second edition.

Roddick, J. F., Al-Jadir, L., Bertossi, L., Dumas, M., Estrella, F., Gregersen, H., Hornsby, K., Lufter, J., Mandreoli, F., Männistö, T., Mayol, E., and Wedemeijer, L. (2000). Evolution and change in data management issues and directions. *ACM SIGMOD Record*, 29(1):21–25.

Roddick, J. F., Craske, N. G., and Richards, T. J. (1994). A taxonomy for schema versioning based on the relational and entity relationship models. *Lecture Notes in Computer Science*, 823:137–148.

Roddick, J. F. (1995). A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393.

Sheth, A. P. and Larson, J. A. (1990). Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236. Also published in/as: Bellcore, TM-STS-016302, Jun.1990.

Sommerville, I. (2001). *Software Engineering*. Addison-Wesley, 6th edition.

Staab, S., Erdmann, M., Mädche, A., and Decker, S. (2000). An extensible approach for modeling ontologies in RDF(S). In *First Workshop on the Semantic Web at the Fourth European Conference on Digital Libraries, Lisbon, Portugal*.

Staab, S. and Mädche, A. (2000). Axioms are objects, too - ontology engineering beyond the modeling of concepts and relations. Technical Report 399, Institut AIFB, Universität Karlsruhe.

Stojanovic, L., Maedche, A., Motik, B., and Stojanovic, N. (2002). User-driven ontology evolution management. In *13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02)*, number 2473 in LNCS, Sigüenza, Spain.

Stuckenschmidt, H. and Klein, M. (2003). Integrity and change in modular ontologies. In *Proceedingso of the 18th International Joint Conference on Artificial Intelligence*, Acapulco, Mexico.

Stuckenschmidt, H. (2000). Using OIL for Intelligent Information Integration. In Benjamins, V. R., Gomez-Perez, A., and Guarino, N., editors, *Proceedings of the Workshop on Applications of Ontologies and Problem-solving Methods, 14th European Conference on Artificial Intelligence ECAI 2000, Berlin, Germany*.

Stuckenschmidt, H. (2003). *Ontology-Based Information Sharing in Weakly-Structure Environments*. PhD thesis, Vrije Universiteit Amsterdam.

Sure, Y., Erdmann, M., Angele, J., Staab, S., Studer, R., and Wenke, D. (2002). OntoEdit: Collaborative ontology development for the Semantic Web. In *First International Semantic Web Conference (ISWC 2002)*, volume 2342 of *LNCS*, pages 221–235. Springer.

Tu, S. W. and Musen, M. A. (1999). A flexible approach to guideline modeling. In *AMIA Annual Symposium*, pages 420–424.

Ventrone, V. and Heiler, S. (1991). Semantic heterogeneity as a result of domain evolution. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 20(4):16–20.

Visser, P. R. S., Jones, D. M., Bench-Capon, T. J. M., and Shave, M. J. R. (1997). An analysis of ontological mismatches: Heterogeneity versus interoperability. In *AAAI 1997 Spring Symposium on Ontological Engineering*, Stanford, USA.

Volz, R., Oberle, D., Staab, S., and Studer, R. (2002). Ontolift prototype. Deliverable D11, EU/IST Project WonderWeb.

Wedemeijer, L. (2001). Defining metrics for conceptual schema evolution. In Balsters, H., de Brock, B., and Conrad, S., editors, *Database Schema Evolution and Meta-Modeling, 9th International Workshop on Foundations of Models and Languages for Data and Objects (FoMLaDO/DEMM 2000)*, volume 2065 of *Lecture Notes in Computer Science*, pages 220–244. Springer.

Wiederhold, G. (1994). An algebra for ontology composition. In *Proceedings of 1994 Monterey Workshop on Formal Methods*, pages 56–61, U.S. Naval Postgraduate School, Monterey CA.

Yan, B., Frank, M., Szekely, P., Neches, R., and Lopez, J. F. (2003). WebScripter: Grass-roots ontology alignment via end-user report creation. In *Proceedings of the 2nd International Semantic Web Conference (ISWC2003)*, Sanibel Island, Florida.

SIKS Dissertation Series

1998

1998-1 Johan van den Akker (CWI)
DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM)
Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD)
A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

1998-4 Dennis Breuker (UM)
Memory versus Search in Games

1998-5 E.W.Oskamp (RUL)
Computerondersteuning bij Straftoemeting

1999

1999-1 Mark Sloof (VU)
Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products

1999-2 Rob Potharst (EUR)
Classification using decision trees and neural nets

1999-3 Don Beal (UM)
The Nature of Minimax Search

1999-4 Jacques Penders (UM)
The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB) *Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems*

1999-6 Niek J.E. Wijngaards (VU)
Re-design of compositional systems

1999-7 David Spelt (UT)
Verification support for object database design

1999-8 Jacques H.J. Lenting (UM)
Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation

2000

2000-1 Frank Niessink (VU)
Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE) *Prototyping of CMS Storage Management*

2000-3 Carolien M.T. Metselaar (UvA)
Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectie

2000-4 Geert de Haan (VU)
ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 Ruud van der Pol (UM)
Knowledge-based Query Formulation in Information Retrieval

2000-6 Rogier van Eijk (UU)
Programming Languages for Agent Communication

2000-7 Niels Peek (UU)
Decision-theoretic Planning of Clinical Patient Management

2000-8 Veerle Coupé (EUR)
Sensitivity Analyis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI)
Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI)
Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI)
Scalable Distributed Data Structures for Database Management

2001

2001-1 Silja Renooij (UU)
Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU)
Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA)
Learning as problem solving

2001-4 Evgueni Smirnov (UM)
Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU)
Processing Structured Hypermedia: A Matter of Style

2001-6 Martijn van Welie (VU)
Task-based User Interface Design

2001-7 Bastiaan Schonhage (VU)
Diva: Architectural Perspectives on Information Visualization

2001-8 Pascal van Eck (VU)
A Compositional Semantic Structure for Multi-Agent Systems Dynamics

2001-9 Pieter Jan 't Hoen (RUL)
Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA)
Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design

2001-11 Tom M. van Engers (VU)
Knowledge Management: The Role of Mental Models in Business Systems Design

2002

2002-01 Nico Lassing (VU)
Architecture-Level Modifiability Analysis

2002-02 Roelof van Zwol (UT)
Modelling and searching web-based document collections

2002-03 Henk Ernst Blok (UT)
Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU)
The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 Radu Serban (VU)
The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL)
Applied legal epistemology; Building a knowledge-based ontology of the legal domain

2002-07 Peter Boncz (CWI)
Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08 Jaap Gordijn (VU)
Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas

2002-09 Willem-Jan van den Heuvel (KUB)
Integrating Modern Business Applications with Objectified Legacy Systems

2002-10 Brian Sheppard (UM)
Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU)
Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 Albrecht Schmidt (UvA)
Processing XML in Database Systems

2002-13 Hongjing Wu (TUE)
A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU)
Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT)
Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU)
The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UvA)
Understanding, Modeling, and Improving Main-Memory Database Performance

2003

2003-01 Heiner Stuckenschmidt (VU)
Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU)
Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD)
Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT)
Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UvA)
Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT)
Development and specification of virtual environments

2003-07 Machiel Jansen (UvA)
Formal Explorations of Knowledge Intensive Tasks

2003-08 Yongping Ran (UM)
Repair Based Scheduling

2003-09 Rens Kortmann (UM)
The resolution of visually guided behaviour

2003-10 Andreas Lincke (UvT)
Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 Simon Keizer (UT)
Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 Roeland Ordelman (UT)
Dutch speech recognition in multimedia information retrieval

2003-13 Jeroen Donkers (UM)
Nosce Hostem - Searching with Opponent Models

2003-14 Stijn Hoppenbrouwers (KUN)
Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 Mathijs de Weerdt (TUD)
Plan Merging in Multi-Agent Systems

2003-16 Menzo Windhouwer (CWI)
Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 David Jansen (UT)
Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 Levente Kocsis (UM)
Learning Search Decisions

2004

2004-01 Virginia Dignum (UU)
A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 Lai Xu (UvT)
Monitoring Multi-party Contracts for E-business

2004-03 Perry Groot (VU)
A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 Chris van Aart (UvA)
Organizational Principles for Multi-Agent Architectures

2004-05 Viara Popova (EUR)
Knowledge discovery and monotonicity

2004-06 Bart-Jan Hommes (TUD)
The Evaluation of Business Process Modeling Techniques

2004-07 Elise Boltjes (UM)
Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 Joop Verbeek (UM)
Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise

2004-09 Martin Caminada (VU)
For the Sake of the Argument; explorations into argument-based reasoning

2004-10 Suzanne Kabel (UvA)
Knowledge-rich indexing of learning-objects

2004-11 Michel Klein (VU)
Change Management for Distributed Ontologies

2004-12 The Duy Bui (UT)
Creating emotions and facial expressions for embodied agents

2004-13 Wojciech Jamroga (UT)
Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 Paul Harrenstein (UU)
Logic in Conflict. Logical Explorations in Strategic Equilibrium